Möglichkeiten und Methoden der Schaltungssynthese

Vom Fachbereich für Mathematik und Informatik
der Technischen Universität Braunschweig

genehmigte Dissertation

zur Erlangung des Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

von

Dipl.-Inform. Peter Blinzer

Eingereicht am 05. Mai 2000
1. Referent: Prof. Dr. Ulrich Golze
2. Referent: Prof. Dr. Rolf Ernst
Mündliche Prüfung am 11. Juli 2000

Kurzfassung

Beim Entwurf digitaler, integrierter Schaltungen ist ihre automatisierte Synthese aus abstrakten Hardwarebeschreibungen textueller oder graphischer Art üblich. Die Synthesemethoden, die in den letzten Jahrzehnten schrittweise durch die Entkopplung von Entwurf und Fertigung, die Abstraktion der Beschreibungen sowie die Automatisierung von Arbeitsschritten gewachsen sind, bieten viele Möglichkeiten zur Beherrschung der unaufhaltsam wachsenden Komplexität von Chips.

Die ständig zunehmende Distanz zwischen einer Spezifikation und der daraus erzeugten Schaltung, ihre automatisierte Überwindung mit Syntheseprogrammen sowie die Differenzierung in unterschiedliche Synthesemethoden erfordern bereits in frühen Entwurfsphasen Entscheidungen, die signifikante Auswirkungen auf die erreichbaren Ergebnisse haben. Der Mangel an Entscheidungskriterien, anhand derer die Auswahl geeigneter Synthesemethoden, die Aufteilung eines Entwurfs in Partitionen für unterschiedliche Methoden sowie die optimale Steuerung eines Syntheseablaufes möglich ist, stellt daher ein bedeutendes Problem dar.

Als Folge dieses Mangels wird in der Entwurfspraxis oft die Register-Transfer-Synthese als Standardansatz auch dann verwendet, wenn abstraktere Ansätze zu besseren Ergebnissen in kürzerer Entwurfszeit führen würden. Andererseits werden die abstrakteren Methoden mit aufwendigen Kunstgriffen teils auch in Bereichen eingesetzt, in denen mit weniger Abstraktion mehr Effizienz im Entwurfsablauf und seinen Ergebnissen erreicht werden könnte. Derartige Fehlentscheidungen sind in ihrem Ausmaß jedoch nur selten konkret erfaßbar, da in der Regel keine alternativen Lösungen mit anderen Synthesemethoden als Referenz angefertigt werden.

In dieser Arbeit wird das Problem fehlender Entscheidungskriterien für die Methoden der Register-Transfer-, der Controller- und der High-Level-Synthese behandelt. Hierzu werden zunächst die Eigenschaften der Synthesemethoden im Detail analysiert und einander gegenübergestellt, woraus sich erste grundlegende Entscheidungskriterien für die Auswahl von Methoden und Entwurfspartitionen ergeben.

Anhand von praktischen Entwürfen, die in vielen Aspekten realen Industrie- und Forschungsprojekten entsprechen und für die alternative Lösungen in den drei betrachteten Synthesemethoden angefertigt wurden, werden die Ergebnisse der Methoden und die Möglichkeiten zu ihrer Beeinflussung untersucht. Auf der Grundlage zahlreicher untereinander variierter Syntheseläufe werden weitere Kriterien für Entwurfsentscheidungen offensichtlich, welche die bereits aus den Eigenschaften der Methoden abgeleiteten Kriterien erweitern und verfeinern.

Durch die in dieser Arbeit entwickelten Entscheidungskriterien können die zu Beginn eines Entwurfs notwendigen Entscheidungen zur Auswahl von Synthese-methoden und zur Aufteilung in Partitionen für unterschiedliche Methoden auf sicherer Grundlage für optimale Ergebnisse getroffen werden. Auswirkungen auf andere Projekte in Bezug auf deren Effizienzsteigerung sind bereits erkennbar.

Vorwort

Wie viele Dinge unserer alltäglichen Welt besitzt auch dieses Schriftwerk neben seinem primären Zweck, eine langjährige Forschungsarbeit und ihre Ergebnisse geordnet vorzustellen, ein Umfeld als Grundlage für die Entstehung.

So wurden mir für diese Arbeit in vielen Diskussionen zahlreiche Anregungen und konstruktive Kritiken von Herrn Prof. Dr. Ulrich Golze gegeben, dem ich hier an erster Stelle danken möchte.

Herrn Prof. Dr. Rolf Ernst gebührt mein Dank für die Übernahme des Co-Referats.

Beeinflußt haben meine Untersuchungen des weiteren die Diskussionen und Projekte mit Herrn Dr. Ulrich Holtmann von Synopsys in Mountain View, Californien, der mir unter anderem die Arbeit mit dem Protocol-Compiler im Rahmen von Beta-Tests ermöglichte. Die FPGA-Projekte von Herrn Dr. Andreas Koch lieferten wichtiges Know-How zur FPGA-Technologie und zu anwendbaren Entwurfsabläufen. Alternative Lösungsansätze für die Schaltungssynthese und den Controllerentwurf steuerte Herr Claude Ackad mit Statemate und Speed-charts bei; zudem half sein kritisches Hinterfragen meiner Ergebnisse mehrfach, vorschnelle Folgerungen und unvollständige Untersuchungen zu vermeiden.

Außerhalb des direkten Umfeldes dieser Arbeit gilt es weiteren Menschen zu danken, in erster Linie meinen Eltern, die durch ihre Erziehung und Förderung mein Studium ermöglichten und mich während meiner Promotion anspornten.

Für die notwendige Ablenkung, welche für die Überarbeitung fehlerhafter Ansätze und die Überwindung von Denkblockaden wichtig war, danke ich meinen Bekannten vom Uni-Sport der TU Braunschweig sowie stellvertretend für alle von mir gern gehörten musikalisch tätigen Menschen den Mitgliedern von A•CANTUS, dem Chor der Architekturstudentinnen und -studenten der TU Braunschweig.

Inhalt

1 Einleitung

Die Entwurfswege für integrierte Schaltungen von den Ideen zu funktionierenden Chips besitzen trotz ihrer regen Benutzung noch große Ähnlichkeit mit den Seewegen über Ozeane in der Anfangszeit der Hochseeschiffahrt. Diese Seewege und ihre Ziele waren einerseits nur grob und unzuverlässig bestimmbar, da es an Navigationshilfen zur Planung und Orientierung mangelte. Andererseits gab es viele Möglichkeiten und Notwendigkeiten zur Auswahl und Korrektur eines Kurses, deren falsche Nutzung die Reise zu einer Odyssee machten.

Die Ursache hierfür liegt in der hohen Dynamik des Entwurfs digitaler, integrierter Schaltungen, die trotz seiner mittlerweile vierzigjährigen Geschichte anhält. Getrieben durch die Möglichkeiten wachsender Integrationsdichten und den steigenden Leistungs- und Komfortbedarf des Marktes auf der einen Seite sowie die Notwendigkeit zur Beherrschung zunehmender Schaltungskomplexität in geringerer Zeit auf der anderen Seite wurden die Methoden des Entwurfs in dieser Geschichte bereits mehrfach grundlegend verändert.

Durch die erfolgreich angewandten Techniken der Entwurfsabstraktion sowie der Entkopplung von Entwurf und Fertigung wurde schrittweise der aktuelle Zustand erreicht, in dem die automatisierte Synthese von Schaltungen aus abstrakten Hardwarebeschreibungen textueller oder graphischer Art üblich ist. Die dabei gewachsene Distanz zwischen einer Entwurfsbeschreibung und der am Ende daraus erzeugten Schaltung wird durch Verbesserungen an Werkzeugen und Arbeitsabläufen bei gleicher Komplexität in immer kürzerer Zeit überbrückt, die Entwurfsgeschwindigkeit also erhöht.

Mit der Zunahme der Weglänge vom Modell zur Schaltung und der Entwurfsgeschwindigkeit gewinnen die Möglichkeiten zur Beeinflussung dieses Weges über die Modellierung, die Synthesemethode und deren Optimierungen an Bedeutung. Die Beeinflussung des Entwurfsweges ist insofern vergleichbar mit der bereits angesprochenen Navigation eines Schiffes. Die Wahl des Anfangskurses hat den größten Einfluß auf die Reisezeit und die Erreichbarkeit des Zieles. Doch auch auf dem Weg selbst bieten sich viele Möglichkeiten und Notwendigkeiten zu seiner Änderung, die bei der Synthese wegen den bisher kaum verfügbaren, unpräzisen Orientierungshilfen aber nur schwer zielgerichtet durchführbar sind.

In dieser Arbeit werden die derzeit besonders bedeutsamen Wege der RTL-, der High-Level- und der Controllersynthese betrachtet. Neben den Unterschieden zwischen ihren Anfangspunkten, Wegstrecken und Problemen werden allgemeine Grundlagen aller drei Methoden betrachtet. Insbesondere wird die Frage nach der für einen Entwurf am besten geeigneten Methode beantwortet, um ihn an sein Ziel zu bringen. Hierzu werden zunächst die Eigenschaften der Entwürfe und ihre Anforderungen an die möglichen Wege eingehend analysiert. Die Untersuchung der Wege auf Hindernisse, die das angestrebte Ziel schwer oder gar nicht erreichen lassen, sowie der Variationen und Optionen, mit denen das gewünschte Ziel möglichst effizient erreicht werden kann, erfolgt an realen Entwürfen.

Neben der Untersuchung der möglichen Wege beschäftigt sich diese Arbeit mit der Bewertung und dem Vergleich der erreichbaren Zielregionen, also gleichsam der Kartographierung der Ozeane und der sie begrenzenden Küsten. Sie stellt damit bisher fehlende Orientierungshilfen zur Verfügung, um den langen Weg von einer Entwurfsaufgabe zu einer Schaltung mit geeigneten Mitteln, in der kürzesten Zeit und mit den bestmöglichen Ergebnissen zu überwinden.

Viele der hierbei behandelten und zu lösenden Probleme entstehen durch die Abstraktion des Entwurfs und den nicht immer deutlich erkennbaren Bezug eines Modells zu seiner Implementierung. Die Abstraktion ermöglicht durch die Vernachlässigung von Details der späteren Schaltung auf der einen Seite einen zügigeren, fehlerfreieren und nicht an bestimmte Technologien gebundenen Ablauf des Entwurfs. Auf der anderen Seite werden durch den wachsenden Abstand einige Eigenschaften und Effekte der späteren Schaltung schwerer kontrollierbar, abschätzbar und nachvollziehbar.

Am Anfang der Geschichte integrierter Schaltungen war der Bezug zwischen einem Entwurf und den aus ihm entstehenden Chips noch offensichtlich. Die einzelnen Schalt- und Verbindungselemente wurden direkt aus Polygonen für die unterschiedlichen Materialien auf ihren Fertigungsebenen zusammengesetzt. Die Belichtungsmasken der Fertigungsebenen wurden aus diesen Entwürfen durch eine maßstabsgerechte Verkleinerung erzeugt, der Entwurf also vollständig auf der Layoutebene durchgeführt. Zur Simulation wurden die Schaltelemente aus den Layoutinformationen extrahiert und mit Analog-Simulatoren wie z.B. SPICE in ihrem Schaltverhalten berechnet. Für Spezialanwendungen wird diese Entwurfsmethode noch heute praktiziert, z.B. für Analogschaltungen oder RAM-Bausteine.

Eine wesentliche Vereinfachung des Entwurfs erfolgte durch die Einführung von abstrakteren Schaltelementen als die im Layout detailliert kontrollierbaren Transistoren, von Standardtransistoren sowie Zellen aus mehreren Transistoren zur Implementierung von logischen Schaltfunktionen der Booleschen Algebra. Die Automatisierung der Plazierung von Schaltelementen sowie ihrer Verdrahtung ermöglichte eine Abstraktion der Entwurfseingabe hin zu Schaltplänen auf der Ebene von Logikgattern. Die Vernachlässigung analoger Effekte und exakter geometrischer Eigenschaften führte zu deutlichen Reduzierungen im Zeitaufwand für die Plazierung und Verdrahtung sowie für die Simulationen. Auf der anderen Seite wurden hierdurch die Möglichkeiten zur Beeinflussung der Verdrahtung, der Zell- und Transistorgeometrien sowie der Schaltungstopologie eingeschränkt. Durch die Verbesserungen im Bereich der Entwurfszeit und -sicherheit wurden die beschränkten Entwurfs- und Optimierungsmöglichkeiten für die meisten digitalen Schaltungen aber mehr als ausgeglichen und die Produktivität signifikant erhöht.

Hardwarebeschreibungssprachen als textuelle, algorithmische Notationen führten zu einer weiteren Abstraktion bei Entwurfseingabe und Simulation. Komplexe Baugruppen sind in ihrer zeitlichen Aktivität, ihrer Funktion sowie ihrer Aufteilung in Register und kombinatorische Logik mit Anweisungsfolgen spezifizierbar, die von CAD-Systemen zur Schaltungssynthese automatisiert auf

eine Zieltechnologie abgebildet werden können. Die Wahl der Schaltungselemente erfolgt dabei automatisch aus Technologiebibliotheken, kann aber auch manuell gesteuert werden. Wichtige Elemente der Entwurfsabstraktion auf die Register-Transfer-Ebene sind die einfache Nutzung synchroner Logik sowie die Beschreibung komplexer kombinatorischer Operationen durch arithmetische Operatoren, Anweisungsfolgen, Selektionen und Vervielfältigungsschleifen.

Zusätzlich zu den bereits beim Entwurf auf der Logikebene in Plazierungs- und Verdrahtungsprogramme sowie Bibliotheken verlagerten Möglichkeiten zur detaillierten Beeinflussung der entstehenden Schaltung wird beim Register-Transfer-Entwurf mit Syntheseprogrammen auf die genaue Kontrolle der Gattertypen und ihrer Verbindungsstruktur verzichtet. Die Mächtigkeit der nutzbaren Operatoren und Anweisungsarten führt im Gegenzug zu einer weiteren Erhöhung der Entwurfsgeschwindigkeit und -sicherheit. Für Simulationen wird mit den zu Ereignisabfolgen in einem Simulationszeitraster abstrahierten Takten und Laufzeitverzögerungen auf Genauigkeit verzichtet und somit der Aufwand an benötigter Rechenzeit verringert.

Die Abstraktion des Schaltungsentwurfs über die Register-Transfer-Ebene hinaus differenziert sich in mehrere Methoden. Diese besitzen unterschiedliche Eigenschaften in Modellierungstechniken und -möglichkeiten und setzen damit verschiedene Abstraktionsschwerpunkte. Während sich die Entwurfsmethoden auf Layout-, Gatter- und RT-Ebene an der technischen Realisierung durch Schaltelemente orientieren, steht bei Methoden oberhalb der RT-Ebene der Bezug zum Anwendungsbereich im Vordergrund. Einige der bekannteren Entwurfsmethoden oberhalb der RT-Ebene sind die Synthese algorithmischer Beschreibungen (High-Level-Synthese), die Controllersynthese, die Datenpfadsynthese sowie Synthese von Prozessoren mit Anwendungen in der digitalen Signalverarbeitung (DSPs) bzw. mit anwendungsabhängigen Befehlen (ASIPs).

Bei realen Entwürfen ist jedoch meist keine direkt zu der vorhandenen Methoden-Differenzierung passende Aufgabenrichtung gegeben. Damit entsteht bereits zu Beginn eines Entwurfs das Problem, die bestmögliche Synthesemethode als Kompromiß zwischen den Anforderungen der Aufgabe und den Eigenschaften bzw. Möglichkeiten der Methode zu wählen. Einerseits ist eine hohe Abstraktion anzustreben, um hohe Sicherheit bei geringer Dauer des Entwurfs zu erreichen, andererseits zu beachten, daß die zur Erfüllung aller Spezifikationen der Aufgabe notwendigen Technologie-Details noch kontrollierbar sind. Gegebenenfalls ist eine Aufteilung der Gesamtaufgabe in Teile vorzunehmen (Partitionierung), für die jeweils andere Methoden angewendet werden.

Bei Entscheidungen für die Partitionierung von Aufgaben und der Auswahl von Methoden besteht allerdings ein Mangel an Entscheidungskriterien, der aus der parallelen, unkoordiniert gewachsenen Differenzierung der Methoden und Werkzeuge oberhalb der Register-Transfer-Ebene resultiert. Wie gut die Auswahl einer Partitionierung und der Methoden die Lösung einer Entwurfsaufgabe erlauben, zeigt sich erst im Entwurfsablauf aus den entstehenden Ergebnissen.

Fehlerhafte Entscheidungen haben zeitaufwendige Korrekturen und Irrwege im Entwurfsablauf zur Folge, die anfangs erwähnten Entwurfs-Odysseen.

Für die Methoden oberhalb der Register-Transfer-Ebene, die auf ihr als gemeinsamer Basis aufsetzen, ergibt sich weiterhin die Frage, auf welchem Weg ein Entwurf optimal von seiner Eingabemethode über den RTL-Knotenpunkt auf seine Zieltechnologie abgebildet werden kann. Auch hierbei unterscheiden sich die Methoden und es fehlt an Orientierungshilfen, um Irrwege zu vermeiden.

In dieser Arbeit bilden die Fragen nach der bestmöglichen Auswahl einer Synthesemethode zu einer Entwurfsaufgabe und der optimalen Abbildung des Entwurfs auf eine Zieltechnologie die zentralen Themen. Für die High-Level-, Controller- und Register-Transfer-Synthese werden diese Fragen beantwortet und die bisher fehlenden Orientierungshilfen entwickelt. Es werden gleichsam Kompaß und Karte bereitgestellt, um im Entwurfsablauf von vornherein einen geeigneten Weg zum Ziel zu finden, statt sich mit der Navigationserfahrung eines bislang durch Flüsse geleiteten Binnenschiffers auf Hohe See begeben zu müssen.

Hierzu wird zunächst auf die allgemeinen Grundlagen des synthesebasierten Schaltungsentwurfs eingegangen (Kapitel 2). Die Festlegung und Erläuterung elementarer Begriffe sowie die Betrachtung verschiedener Sichtweisen auf die Abläufe und die durch sie beeinflußten Eigenschaften klären einleitend die Frage, was Synthese im Schaltungsentwurf überhaupt ist und welche grundlegenden Verfahren und Methoden existieren. Die Sprachen und Werkzeuge, die im Bereich dieser Arbeit für Entwurfseingabe, -simulation und -synthese von Bedeutung sind, werden in der grundsätzlichen Anwendung, der Einbettung in Entwurfsabläufe und dem Bezug zur umgebenden Synthesewelt vorgestellt. Zu den in dieser Arbeit benutzten Hardwarebeschreibungssprachen Verilog und Protocol-Compiler-HDL werden vertiefende Einzel- und Vergleichsbetrachtungen vorgenommen. Beendet wird der Grundlagenbereich durch den Vergleich von Bewertungsmaßen für FPGAs und ASICs in der Übertragbarkeit auf die jeweils andere Technologie, welche eine wichtige Voraussetzung für die technologieunabhängige Nutzung der Bewertungen und der daraus entwickelten Kriterien darstellt.

In Erweiterung dieser Grundlagen wird in Kapitel 3 die RTL-Synthese mit Verilog intensiver betrachtet. Hierbei stehen die Modellierungskonstrukte, ihre Möglichkeiten und ihre Korrespondenzen mit Hardware-Implementierungen im Vordergrund. Zur Modellierung kombinatorischer und sequentieller Logik werden dabei einfache Richtlinien gegeben. Möglichkeiten zur fehlerhaften Modellierung sowie ihre Auswirkungen werden ebenfalls erörtert. Auf das Syntheseverfahren selbst wird nur soweit eingegangen, wie es für die Anwendung nötig ist. Unter anderem werden die sich in Modellierung und Synthese bietenden Optimierungen miteinander verglichen.

Aufbauend auf der Modellierung und Synthese mit RTL-Verilog stellt Kapitel 4 die High-Level-Modellierung mit Verilog und deren Synthese vor. Dabei werden zunächst die Unterschiede zur RTL-Modellierung und insbesondere die bei der High-Level-Synthese zu beachteten Einschränkungen herausgehoben. Die FSMD-Zielarchitektur, die unterschiedlichen Ein-/Ausgabesynchronisationen sowie die

allgemeine Arbeitweise von Scheduling und Allokation sind weitere Schwerpunkte dieser Betrachtungen.

Kapitel 5 vertieft die Controllersynthese mit dem Protocol-Compiler als letzte der betrachteten Methoden. Die im Grundlagenbereich nicht vorgestellten Teile einer Protokolldefinition werden eingeführt und die Optionen der Controllersynthese bezüglich FSM-Strukturen und Zustandscodierungen erläutert. Dabei wird die Implementierung von Protokollelementen in Hardware, z.B. bei Zählern, unter den Aspekten der Ressourceneinsparung und der Zustandskomplexität näher vorgestellt.

Die Vergleiche der drei Synthesemethoden in ihren Eigenschaften und ihren Ergebnissen anhand verschiedener Entwürfe sowie die daraus konstruierten Kriterien zur Auswahl der jeweils bestmöglichen Methode und ihrer optimalen Anwendung bilden den Kernbereich dieser Arbeit, Kapitel 6. Die Analyse der Eigenschaften der Methoden anhand ihrer Entwurfseingaben ist der Ausgangspunkt der Untersuchungen, die durch empirische Betrachtungen an mehreren Entwürfen weitergeführt werden. Auf der Basis der Eigenschaften von Aufgaben und Methoden sowie der erzielbaren Ergebnisse wird eine Entwurfsklassifizierung erarbeitet, die eine Auswahl bestmöglich geeigneter Synthesemethoden zu einer Entwurfsaufgabe erlaubt. Die in den zahlreichen Einzelversuchen bestimmten Ergebnisräume charakterisieren außerdem die Auswirkungen verschiedener Synthesevariationen und ermöglichen so den Rückschluß auf die jeweils optimalen Synthesewege für die einzelnen Methoden.

Kapitel 7 faßt die Ergebnisse dieser Forschungsarbeiten zusammen und gibt Einblicke in bereits vorhandene oder absehbare Auswirkungen auf andere Projekte.

Die in dieser Arbeit vorgestellten Erkenntnisse erlauben für drei Methoden bessere Implementierungen in einer Zieltechnologie zu erreichen. Hierzu wurden praktische Entwürfe in zahlreichen Variationen untersucht, wofür mehrere Monate Rechenzeit benötigt wurden. Die Vervollständigung um weitere Entwürfe und Methoden, welche die Allgemeingültigkeit der getroffenen Aussagen untermauern und vielleicht weitere Kriterien zu Tage fördern wird ebenso ihren Tribut in Arbeitsaufwand und Rechenzeit fordern, durch den daraus erwachsenden Nutzen an verringerter Entwurfsdauer und verbesserter Entwurfsqualität aber wie bei dieser Arbeit gerechtfertigt werden.

Im Gegensatz zu den bereits aufgezeigten Ähnlichkeiten mit der Seefahrt sind für die notwendigen Untersuchungen der Entwurfswege jedoch weitaus geringere Gefahren und Kosten in Kauf zu nehmen, als sie der römische Kaiser Pompejus (106 - 48 v.Chr.) für seine Seeflotte in Worte faßte:

Navigare necesse est, vivere non est necesse.

Die Seefahrt ist notwendig, das Leben einzelner nicht.

2 Allgemeine Grundlagen zur Schaltungssynthese

In diesem Kapitel wird in das Themengebiet der Arbeit weiter eingeführt. Neben der groben Beschreibung und Eingrenzung des Themengebietes werden grundlegende Begriffe, Verfahren und Probleme vorgestellt.

Im Rahmen von Unterkapiteln wird zunächst die Schaltungssynthese in ihrer allgemeinen Definition, ihrem begrifflichem Umfeld sowie der eingeschränkten Betrachtung dieser Arbeit behandelt.

Anschließend werden Werkzeuge und Sprachen vorgestellt, die im Umfeld der Schaltungssynthese von Bedeutung sind. Auf die verwendeten Werkzeuge und Sprachen wird soweit vertiefend eingegangen, wie es für das Verständnis ihrer Anwendungsweise notwendig ist.

2.1 Synthese im digitalen Schaltungsentwurf

In der allgemeinen, sprachlichen Definition des Begriffs Synthese bedeutet dieser die Zusammensetzung einzelner Teile zu einem Ganzen oder auch die Bildung von etwas Komplexen aus etwas Einfachem. Im Schaltungsentwurf bezeichnet er entsprechend die Zusammensetzung von Komponenten zu einer Schaltung mit bestimmten, geforderten Eigenschaften [Gajski88].

Diese erste, noch sehr allgemeine Erklärung für die Schaltungssynthese ist prinzipiell für alle Formen von digitalen, analogen, integrierten und diskreten Schaltungen gültig. In dieser Arbeit wird aber nur der Bereich der integrierten, digitalen Schaltungen betrachtet werden, wodurch z.B. die Eigenschaften eines Entwurfes vergleichsweise überschaubar bleiben.

Das bekannteste Modell zur Einordnung der Entwurfseigenschaften von integrierten Schaltungen in einen Zusammenhang ist das Y-Diagramm. Dieses wurde 1983 von Daniel Gajski und Robert Kuhn vorgestellt [GajKuh83] und zu der Form in Bild 2.1 weiterentwickelt [MicLau92]. Es verwendet eine dreigeteilte Sicht auf Entwürfe aus Struktur, Geometrie und Verhalten, die im Diagramm drei radiale Geraden bilden. Im Treffpunkt der drei Geraden ist die reale Schaltung angesiedelt, die alle drei Perspektiven mit maximaler Detailliertheit vereint. Mit wachsendem Abstand vom Mittelpunkt steigt der Abstraktionsgrad der Entwurfsdarstellung (Schaltkreis-, Logik-, Register-Transfer-, Algorithmus-, und Systemebene). Im Rahmen dieses Modells bezeichnet die Synthese einer Schaltung den Übergang von einem Verhalten zu einer Struktur, beispielsweise von Boolescher Logik zu Gattern durch deren Zusammensetzung.

Erläuterung 2.1 Das *Verhalten* eines Entwurfs ist seine Ausgabereaktion auf angelegte Eingabedaten und die fortschreitende Zeit. Es ist eine Sicht, welche sich auf die Schnittstellen eines Entwurfs zur Umgebung und die Abhängigkeiten ihrer Daten untereinander und vom Zeitverlauf konzentriert. Die *Struktur* eines Entwurfs erfaßt die Komponenten einer Schaltung, die durch das Entwurfsumfeld (z.B. die Schaltungstechnologie) vorgegeben sind, und ihre Verbindungen. Unter dem Begriff *Geometrie* werden schließlich die Informationen über Fläche, Form und Lage von Entwurfsteilen eingeordnet.

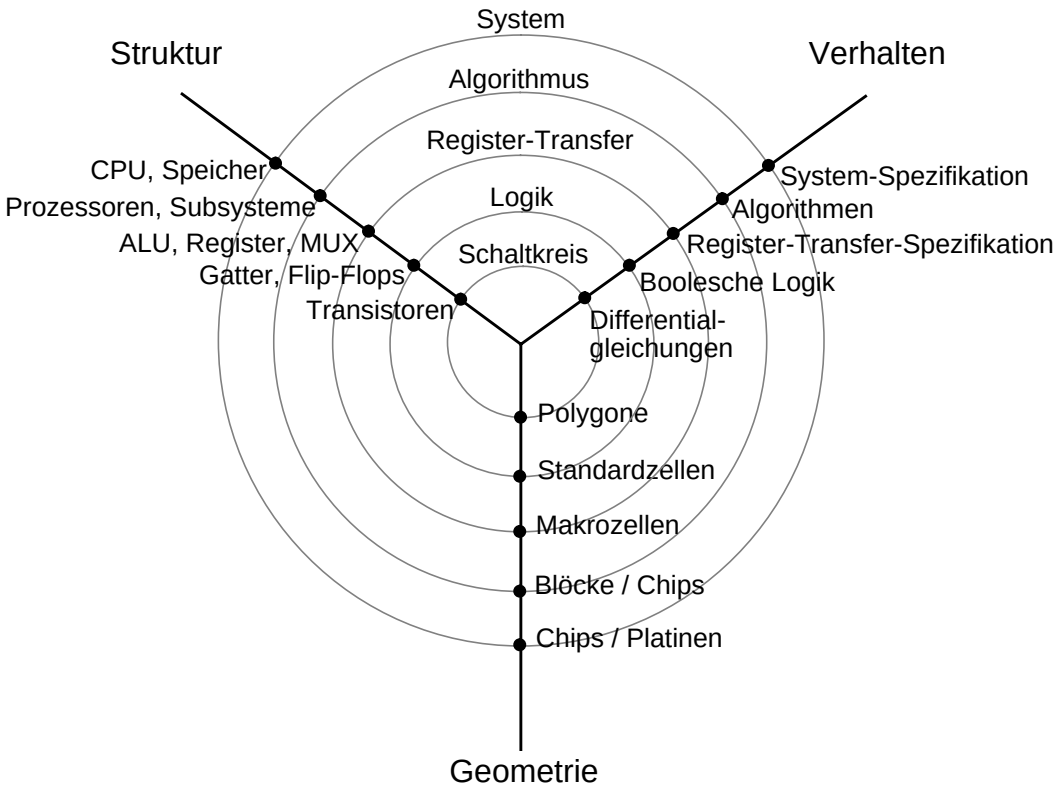


Bild 2.1: Die dreigeteilte Sicht auf Entwurfsebenen im Y-Diagramm

Trotz seiner weiten Verbreitung als Gedankenmodell besitzt das Y-Diagramm deutliche Mängel bei der Einordnung von Entwurfseigenschaften in der Praxis. So werden zeitliche Aspekte nur implizit über die Verhaltensperspektive erfaßt, obwohl sie in Wirklichkeit für Pfad- und Komponentenlaufzeiten eng an die Struktur gebunden sind.

Die Unterscheidungskriterien zwischen Abstraktionsebenen sind bei näherer Betrachtung auch nicht widerspruchsfrei. So entstehen beispielsweise Register aus Flip-Flops technisch durch reguläre Vervielfältigung, ohne daß sich dabei Schnittstellen in irgendeiner Hinsicht außer der Bitbreite ändern. Register sind damit eine Hierarchiestufe des Entwurfs, abstrahieren aber keine Flip-Flop-Eigenschaften. Im Vergleich dazu abstrahieren ALUs zwar Eigenschaften von Gattern, allerdings durch den Übergang von Boolescher Logik zu arithmetischen Operationen. Strukturell sind sie nur Elemente einer Entwurfshierarchie, die auf einfacheren Elementen aufbaut und dabei deren Schnittstellen- und Verbindungskonzepte vollständig beibehält. In der Verhaltensperspektive erfolgt der Übergang von der Logikebene zur Register-Transfer-Ebene durch die Einführung von Datenspeicherung und getakteter Zeit zusätzlich zu Boolescher Logik und kontinuierlicher Zeit. Register-Transfer-Spezifikationen sind allein mit Boolescher Logik nicht konstruierbar, womit dieser Übergang ebenfalls keine Abstraktion von Eigenschaften darstellt.

Zudem ist die strikte, nicht abgestufte Einteilung von Eigenschaften in diese drei getrennten Perspektiven oft mehr verwirrend als hilfreich, wenn ein Entwurf entsprechend der im Diagramm genannten Elemente der Abstraktionsebenen einer der drei Perspektiven zugeordnet werden soll. Entwurfsdarstellungen kombinieren im Regelfall Elemente der drei Perspektiven und selbst die als Beispiele genannten Entwurfselemente umfassen mehr als nur eine dieser Perspektiven. Die sich aus dem Y-Diagramm ergebende Bezeichnung eines Entwurfs aus Flip-Flops und Gattern als *Strukturbeschreibung* ist sehr irreführend, da diese Komponenten und die darauf aufbauenden Entwürfe ein definiertes Verhalten besitzen, das im Entwurf bewußt benutzt wird. Die Bezeichnung *Verhaltensbeschreibung* für Register-Transfer-Spezifikationen ist ebenfalls unangemessen, da diese Spezifikationen genau zwischen Registern und kombinatorischer Logik unterscheiden und damit Strukturen von Komponenten eindeutig festlegen. Ebenso wie das Y-Diagramm haben sich aber auch die daraus resultierenden Begriffe als Schlagworte weit verbreitet. Aufgrund der mit ihnen verbundenen Widersprüche werden sie in dieser Arbeit bewußt vermieden.

Ein alternatives Modell zum Y-Diagramm ordnet Entwurfsbeschreibungen in mehrdimensionale Entwurfsräume ein, die von verschiedenen Eigenschaften des Entwurfs aufgespannt werden und nach Detailliertheit der Betrachtung abgestuft sind [EckHof92]. Die Dimensionen dieser Entwurfsräume können voneinander unabhängig sein, müssen dies aber nicht als Voraussetzung erfüllen. Bild 2.2 zeigt einen Entwurfsraum, der zur Vereinfachung der Darstellung auf drei Dimensionen beschränkt ist und die Darstellung von Zeit, Daten und Datenverarbeitung erfaßt.

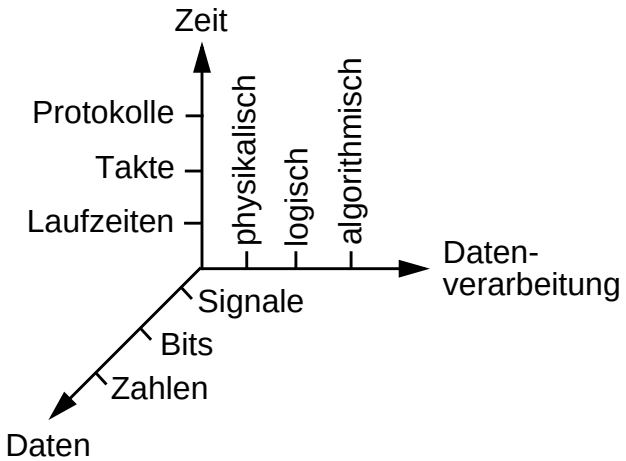


Bild 2.2: Entwurfsraum mit drei Dimensionen

Weitere Eigenschaften, die als Dimensionen eine Erweiterung dieses einfachen Entwurfsraumes ermöglichen, sind die Hierarchieebene der Entwurfselemente (Logik-, RT-, Algorithmus- und Systemebene), ihre Auflösung der Struktur (Polygone der Chip-Masken, Transistoren, Gatter, Register und kombinatorische Logik oder Funktionsblöcke) sowie die Beschreibungsart (Differentialgleichungen, Logikterme, Algorithmen oder Ablaufdiagramme).

Ein wichtiger Unterschied dieser mehrdimensionalen Darstellungsweise zum Y-Diagramm ist die Erfassung eines Entwurfs über die Kombination mehrerer seiner Eigenschaften anstelle der Abgrenzung von Eigenschaften gegeneinander. Entwurfsbeschreibungen werden damit nicht nach Verhalten, Struktur oder Geometrie unterschieden, sondern je nach Kombination von Eigenschaften in Teilbereiche des Entwurfsraumes eingeordnet. Eine genaue Zuordnung zu einzelnen Ausprägungen von Eigenschaften, wie sie das Y-Diagramm für ganze Entwürfe suggeriert, erfolgt nur für elementare Entwurfskomponenten. Da auch bei der schwerpunktmäßigen Betrachtung einzelner Eigenschaften in jeder Form der Darstellung mehrere Eigenschaften berücksichtigt werden, sind derartige Betrachtungen beispielsweise (mehr) *verhaltensorientiert* oder (mehr) *strukturorientiert*, aber nicht auf diese Eigenschaften beschränkt.

Erläuterung 2.2 Die *Synthese* verfeinert in einer räumlichen Entwurfsdarstellung eine oder mehrere Eigenschaften einer Entwurfsbeschreibung, indem sie diese Eigenschaften durch Elemente einer detaillierteren Beschreibungsform nachbildet. Hierdurch entsteht eine neue Entwurfsbeschreibung, deren Bereichsgrenzen im Entwurfsraum in Richtung größerer Detailliertheit verschoben sind.

Bei mehr als drei Dimensionen besitzt das räumliche Modell jedoch den Nachteil, die menschliche Vorstellung und graphische Repräsentationsformen zu überfordern. Als brauchbarste Alternative zur teils verwirrenden und fehlerhaft vereinfachenden Darstellung des Y-Diagramms wird es dennoch als Grundlage für den weiteren Verlauf dieser Arbeit verwendet.

Je nach Ausgangsbeschreibung und Ergebnis von Syntheseabläufen werden diese unterschiedlich benannt. Die Benennung der Syntheseabläufe über die Hierarchieebene von Entwurfselementen ist die Grundlage für ihre verbreiteten Bezeichnungen, die aufgrund historischer Entwicklungen jedoch nicht einheitlich vom Ausgangspunkt oder Ziel der Synthese abgeleitet werden.

Erläuterung 2.3 Die *Layoutsynthese* bestimmt eine geometrische Anordnung von Elementen der Fertigungsschnittstelle (z.B. Polygone oder Logikzellen und Verdrahtungssegmente) ausgehend von einer gröberen Beschreibungsform, wie der Verbindungsstruktur von Schaltelementen.

Erläuterung 2.4 Die *Logiksynthese* bestimmt die Bauteile der Logikebene und ihre Verbindungen, die zur Realisierung gegebener boolescher Funktionen oder Register benötigt werden. Die Bauteile können allgemein übliche Elemente der Logikebene sein, wie NAND-Gatter und D-Flip-Flops (*generische Logik*) oder spezielle Elemente einer Schaltungstechnologie. Die Abbildung generischer Logik auf eine Schaltungstechnologie wird auch oft getrennt betrachtet und als *Technology-Mapping* bezeichnet. Das Ergebnis ist eine Verbindungsstruktur von Schaltelementen der Logikebene, eine *Gatternetzliste*.

Erläuterung 2.5 Bei der *RTL-Synthese* werden mit kombinatorischen und getakteten Anweisungen beschriebene Register-Transfers in Speicherelemente und Boolesche Funktionen zerlegt und in eine Gatternetzliste überführt. Hierbei wird

die Logiksynthese und ein Technology-Mapping für kombinatorische Logik und Register ausgeführt. Die Notation der Anweisungen kann je nach Eingabesprache textuell oder graphisch erfolgen.

Erläuterung 2.6 Bei der *Synthese algorithmischer Beschreibungen* werden im Stil imperativer Algorithmen beschriebene Berechnungen und Abläufe entsprechend ihrer Operationen und Datenabhängigkeiten in ein getaktetes Zeitschema und eine Verarbeitungsarchitektur eingeordnet. Das Zeitschema und die Architektur entsprechen Register-Transfers, so daß die RTL-Synthese den weiteren Weg zur Fertigung bereitstellt. Die Synthese algorithmischer Beschreibungen wird auch *High-Level-Synthese* genannt.

Bei der RTL-Synthese beginnt beispielsweise ein typischer Syntheseablauf mit einer Beschreibung des Entwurfs in Sprachen, die das Verhalten in Mischformen aus Logiktermen und prozeduralen Anweisungen erfassen. Die Elemente dieser Sprachen können Entwurfshierarchien von der Logikebene an aufwärts bilden und so die Struktur aus kombinatorischen Logiken und Registern bis hin zu einzelnen Gattern und Flip-Flops auflösen. Die Basisebene eines Entwurfes kann aber auch bis auf die Ebene von Register-Transfers angehoben sein. Die Zeit wird in Form von Laufzeiten oder Takten dargestellt.

Über mehrere Zerlegungsschritte ermittelt die RTL-Synthese zunächst, welche Register-Transfer-Komponenten direkt an das Technology-Mapping übergeben werden können. Die übrigen RT-Komponenten werden mittels Logiksynthese ihrer Booleschen Funktionen auf das Technology-Mapping vorbereitet, welches als Ergebnis eine Gatternetzliste der gewählten Zieltechnologie liefert. Die Elemente dieser Netzliste liegen auf der Logikebene der Entwurfshierarchie und lösen sie in eine Verbindungsstruktur aus Gattern, Flip-Flops und anderen Bauteilen der Zieltechnologie auf. Das Verhalten dieser einzelnen Bauteile kann mit denselben Sprachen dargestellt werden, wie die Eingangsbeschreibung, also durch eine Mischform aus Logiktermen und Anweisungen. Die Zeit wird weiterhin durch Laufzeiten und Takte beschrieben. Sie ist jedoch keine Vorgabe, wie bei der ursprünglichen Beschreibung, sondern eine Folge der Zieltechnologie und ihrer Elemente.

Bei eingeschränkter Betrachtung eines Entwurfes auf die Hierarchieebene der Entwurfselemente und die Darstellung der Zeit läßt sich die RTL-Synthese in einem zweidimensionalen Ausschnitt des Entwurfsraumes veranschaulichen (Bild 2.3).

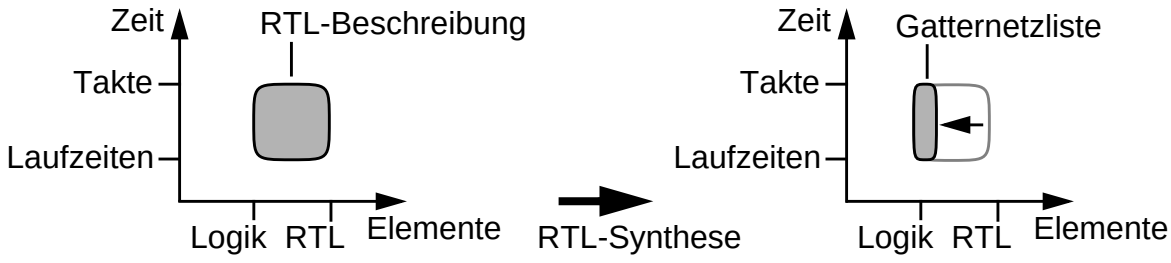


Bild 2.3: Verschiebung der Grenzen des Entwurfsbereiches bei der RTL-Synthese

Es wird hierbei die obere Grenze für die Hierarchieebene der Entwurfselemente hin zur Logikebene verschoben und der Gesamtentwurf damit verfeinert, während die Zeit weiterhin durch Takte und Laufzeiten dargestellt wird.

2.2 Überblick über Werkzeuge und Verfahren

Bei dem Entwurf einer Schaltung mit Synthese kommen verschiedene Formen von Entwurfsbeschreibungen und CAD-Werkzeugen zum Einsatz, die hier im Überblick vorgestellt werden (Bild 2.4).

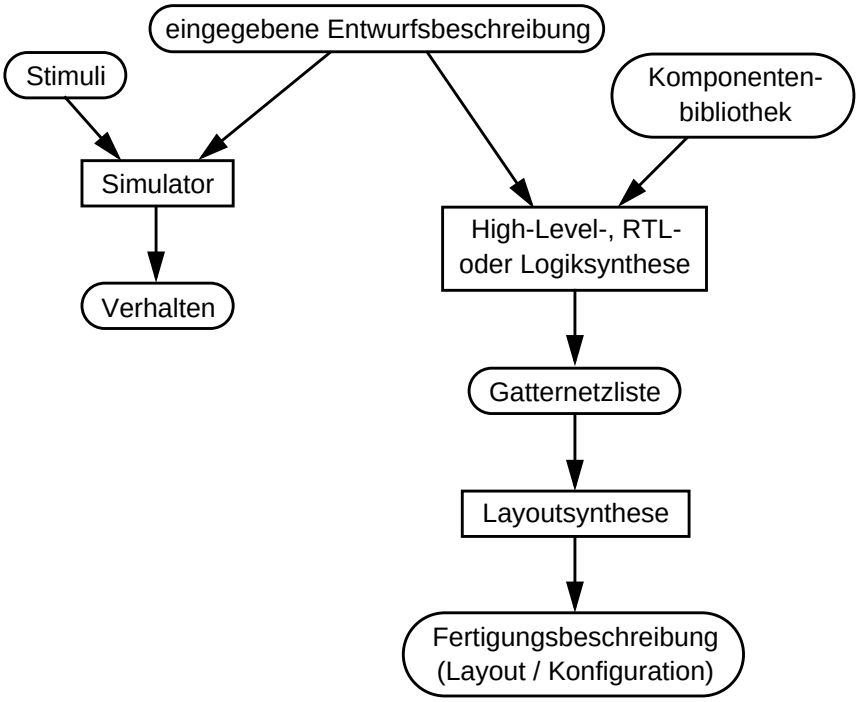


Bild 2.4: Grober Aufbau eines synthesebasierten Entwurfsablaufs

Die Eingabe einer Entwurfsbeschreibung erfolgt in Sprachen, die fest definiert und maschinell verarbeitbar sind. Diese können an textuelle Programmiersprachen angelehnt sein oder aus graphischen Konstrukten bestehen. Diese Sprachen besitzen Elemente zur Beschreibung zeitlichen Verhaltens, auf die Darstellung von digitalen Signalen zugeschnittene Datentypen und in der Regel auch Konstrukte zur getrennten Erfassung parallel aktiver Schaltungsteile. Aufgrund ihres inhärenten Hardwarebezugs werden diese Sprachen als *Hardware Description Languages* (HDLs, *Hardwarebeschreibungssprachen*) bezeichnet.

Der eingegebene Entwurf wird mit Hilfe von *Simulatoren* in seinem reaktiven Verhalten auf *Stimuli* und die fortschreitende Simulationszeit untersucht. Dies erlaubt die Prüfung auf Vollständigkeit und Korrektheit.

Je nach Eingangsbeschreibung wird eine Gatternetzliste mit der High-Level-, RTL- oder Logiksynthese bestimmt. Neben diesen drei Verfahrensbezeichnungen gibt es noch weitere, die eine Differenzierung der Syntheseabläufe über die Hierarchie der Entwurfselemente hinaus erlauben. So gibt es unter anderem noch

die Controllersynthese und die Datenpfadsynthese. Letztere kann sowohl bei RTL-Beschreibungen als auch bei algorithmischen Beschreibungen ansetzen. Für die High-Level-Synthese hat sich infolge des Y-Diagramms ebenfalls die Bezeichnung Verhaltenssynthese verbreitet, die hier aber nicht weiter benutzt wird.

Die Layoutsynthese schließt diesen Entwurfsablauf zur Fertigung bzw. Programmierung von Chips hin ab.

Auf die Sprachen, Simulatoren, Synthesesysteme und -verfahren wird in den folgenden Unterabschnitten näher eingegangen. Den für das Verständnis dieser Arbeit wichtigen Sprachen, Werkzeugen und Verfahren wird dabei besondere Aufmerksamkeit geschenkt.

2.2.1 Sprachen

Hardwarebeschreibungssprachen sind Notationen, welche im Schaltungsentwurf die Darstellung bestimmter Eigenschaften einer Schaltung erlauben. Von den zahlreichen in den letzten 30 Jahren entstandenen HDLs [Mermet93][MicLau92] haben jedoch nur vergleichsweise wenige in der praktischen Anwendung überlebt und nur ein sehr kleiner Teil ist für diese Arbeit von Bedeutung.

Im folgenden werden nur HDLs betrachtet, die im synthesebasierten Entwurf integrierter Schaltungen verbreitet sind und ein Entwurfsverhalten über Register-Transfers oder algorithmische Beschreibungen darstellen können. Spezialsprachen für Modulgeneratoren, die jeweils nur arithmetische Operatoren, Speicherblöcke oder andere funktional eingeschränkte Entwürfe beschreiben, gehören nicht dazu, sind aber gleichwohl von Bedeutung.

Die derzeit gängigsten HDLs für die RT- und die Algorithmus-Ebene sind *VHDL* und *Verilog*, die 1996 einer Studie der *Association of Design Automation Companies* (EDAC) zufolge weltweit insgesamt ähnliche Marktanteile besaßen [CADrep96]. Seither ist von der EDAC eine Schwerpunktverlagerung zu Verilog beobachtet worden, aber damit einhergehend wurden durch Programme für Simulation und Synthese diese textuellen, genormten Sprachen verstärkt gemeinsam unterstützt. Die klare Trennungslinie zwischen den beiden Sprachwelten, die über sehr ähnliche Fähigkeiten und Beschreibungsmöglichkeiten verfügen, wurde auf diese Weise verwischt.

Neben den rein textuellen, werkzeugunabhängigen Sprachen Verilog und VHDL, die im HDL-basierten Entwurf ab der RT-Ebene aufwärts dominant sind, haben sich Entwurfswerkzeuge mit Mischformen aus graphischen und textuellen Entwurfselementen etabliert. Die HDLs dieser Werkzeuge sind geistiges Eigentum des jeweiligen Herstellers und die Entwürfe sind nur indirekt über generierbaren Verilog- bzw. VHDL-Code zwischen den Werkzeugen für Entwurf, Simulation und Synthese austauschbar. Derartige Entwurfswerkzeuge sind also eher als Hilfen für den abstrahierten Verilog- bzw. VHDL-Entwurf zu sehen und nicht als Ersatz für diese beiden klassischen HDLs.

Durch die zeitliche Bindung der vorgegebenen Entwurfselemente an einen Ausführungstakt sind diese HDLs im wesentlichen auf die RT-Ebene beschränkt. Die Definition eigener Entwurfselemente über Verilog- bzw. VHDL-Code erlaubt

zwar die Einbeziehung der Logikebene in solche Entwürfe, verläßt dabei aber den Sprachbereich dieser Entwurfswerkzeuge. So sind im Gegensatz zu den vorgegebenen Elementen innerhalb eigener Verilog- bzw. VHDL-Definitionen keine schrittweisen Simulationen möglich und auch andere Fähigkeiten der Werkzeuge zur Fehlervermeidung oder -suche nicht anwendbar. Zu diesen Werkzeugen bzw. Sprachen gehören:

- Statemate von i-Logix
- Renoir von Mentor Graphics
- Visual HDL von Summit
- Protocol-Compiler von Synopsys

Typische Entwurfselemente der ersten drei Werkzeuge sind hierarchische Zustandsgraphen (statecharts), Anweisungsblöcke (block charts, activity charts), Wertetabellen (truth tables) und Flußdiagramme (flow charts). Der Protocol-Compiler verwendet im Gegensatz dazu eine an Grammatiken für formale Sprachen angelehnte Notation, die für sein Hauptanwendungsgebiet des Controllerentwurfs in der Datenkommunikation besonders geeignet ist.

Die grundsätzliche Fähigkeit zur Lösung von Problemstellungen auf Register-Transfer-Ebene ist bei jeder der vorgenannten vier HDLs vorhanden. Je nach Anwendungsgebiet und Spezifikation der Problemstellung erweist sich zwar die eine oder die andere Eingabemethode als günstiger, aber der auffälligste Vorteil dieser graphisch unterstützten Eingabemethoden liegt in der Abstraktion des Entwurfs endlicher Automaten (*finite state machines*, FSMs). Der Entwickler wird von der zeitraubenden und fehleranfälligen Aufgabe befreit, sich um alle Zustände, deren Übergänge und Codierungen selbst explizit kümmern zu müssen. Stattdessen sind die Zustände in graphische Elemente abstrahiert, ihre Übergänge je nach Eingabevariante teils implizit in der Entwurfsstruktur verankert und die Codierungen automatisch zugewiesen.

Im folgenden wird allerdings nur der Protocol-Compiler weiter betrachtet. Eine Berücksichtigung aller Werkzeuge würde den Rahmen dieser Arbeit zum einen bei weitem sprengen, zum anderen ist die grundsätzliche Möglichkeit des abstrakten Entwurfs endlicher Automaten ebenfalls in ausreichender Weise am Protocol-Compiler demonstrierbar. Durch die Teilnahme am Beta-Test des Protocol-Compilers und direkten Austausch von Entwurfserfahrungen mit dem Entwicklungsteam des Werkzeugs war es außerdem möglich, das Werkzeug bestmöglich zu nutzen und irreführende Ergebnisse aufgrund mangelhafter Beschreibungsvarianten zu vermeiden.

Unabhängig von der jeweils benutzten Notationsform besitzen alle genannten HDLs mehrere Gemeinsamkeiten und Unterscheidungsmerkmale. Gemeinsamkeiten sind gegeben durch:

- die Definition von Syntax und Ausführungsverhalten
- Datentypen zur Modellierung digitaler Signale
- Sprachelemente zur Modellierung der Zeit

- die Beschreibbarkeit paralleler Vorgänge

In bestimmten Merkmalskombinationen unterscheiden sich die Sprachen aber deutlich voneinander. So gibt es verschiedene Ausprägungen für wichtige Merkmale:

- in der Auflösung der Entwurfsstruktur
- bei der Notationsform, die graphisch oder textuell sein kann
- in der Sprachfestlegung durch Normen oder Hersteller
- im Zeitschema, das beliebig modellierbar oder synchron zu einem Takt ist
- in den Hierarchieebenen der Entwurfselemente

Die allgemeinen Eigenschaften und charakteristischen Merkmale der Sprachen werden im folgenden für VHDL, Verilog und die Protocol-Compiler-HDL getrennt vorgestellt.

VHDL: Die Hardwarebeschreibungssprache VHDL wurde 1983 auf Initiative des amerikanischen Verteidigungsministeriums als *VHSIC Hardware Description Language* des Forschungsprogramms *Very-High Speed Integrated Circuits* [Mermet93] entwickelt. Als Grundlage für die Syntax und viele semantische Eigenschaften wurde die textuelle Programmiersprache ADA verwendet. Dies resultierte in einem modularen Konzept mit Trennung von Interface und Implementierung, einer integrierten Verwaltung von Entwurfssichten und Bibliotheken sowie einer strengen Typkontrolle von Daten. Die Normung der Sprache erfolgte 1987 mit dem IEEE Std 1076-1987 und der späteren Überarbeitung IEEE Std 1076-1993 [VHDL94].

In VHDL sind Beschreibungen sowohl struktur- als auch verhaltensorientiert möglich. Als Sprachmittel stehen hierfür Instanzen von Beschreibungen, ihre Verbindung durch Signale und parallel arbeitende Prozesse von Anweisungen zur Verfügung.

Durch die Sprachnorm selbst werden sehr wenige Datentypen vorgegeben, die für Bitsignale nur die Wertigkeiten 0 und 1 zulassen. Signale mit hochohmigem oder undefiniertem Zustand oder die Beschreibung verschiedener Signalstärken zur Konstruktion verdrahteter Logik sind direkt nicht möglich. Durch genormte VHDL-Bibliotheken (IEEE Std 1164) werden aber Datentypen für digitale Signale bereitgestellt, welche die Darstellung von 0-Bits und 1-Bits mit Treiberstärken, hochohmigen Signalen und undefinierten Signalen durch eine Auflösung in 9 Wertigkeiten erlauben. Hierdurch werden Beschreibungen auf der Logikebene möglich. Neben diesen Datentypen für Schaltungsbeschreibungen sind außerdem Typen für ganze Zahlen (INTEGER) und Realzahlen (REAL) vorgegeben. Aus allen definierten Typen können mit Feldern und Records Datentypen selbst definiert werden. In Kombination mit den Anweisungen in Prozeßblöcken und Instanzen sind damit auch die RT- und die Algorithmus-Ebene durch VHDL abgedeckt. Es muß aber beachtet werden, daß die Möglichkeit zur Nutzung selbstdefinierter Datentypen zwar durch die Sprache und Simulationsprogramme erlaubt ist, Synthesesysteme dies aber nur eingeschränkt oder gar nicht unterstützen.

In VHDL ist ein Zeitmodell für Simulationen verankert. Dieses kann zur Darstellung des Zeitverhaltens von Signalen benutzt werden. Neben der Laufzeitverzögerung (*transport delay*) von Signalen ist auch die Trägheitsverzögerung (*inertial delay*) durch Sprachelemente darstellbar. In Kombination mit den Prozeßaktivierungen über Sensitivitätslisten kann damit synchrone und asynchrone Logik beschrieben werden. Synthesesysteme ignorieren solche Verzögerungen jedoch gänzlich und leiten das reaktive Zeitverhalten ausschließlich aus den Prozeßsensitivitätslisten ab.

Verilog: Verilog ist eine textuelle Hardwarebeschreibungssprache, die 1984 in Anlehnung an die Programmiersprache C zusammen mit einem Simulator bei der Firma Gateway Design Automation entwickelt wurde. Die Sprache war lange Zeit an den Hersteller und das Simulationswerkzeug gebunden. Erst 1991 wurde aufgrund des stark wachsenden Marktanteils von VHDL als direkter Konkurrenz zu Verilog die Sprache freigegeben und damit herstellerunabhängig. Die Normung erfolgte 1995 mit dem IEEE Std 1364-1995 [VerStd96]. Die Sprache verfügt aufgrund ihrer Abstammung von C über ein im Vergleich zu VHDL sehr einfaches Modulkonzept, eine sehr grobe Typkontrolle von Daten, automatische Typkonvertierungen und keine Mechanismen zur Verwaltung von Sichten und Bibliotheken. Doch gerade wegen der Einfachheit des Sprachkonzeptes von Verilog, das beim Entwurf kaum Verwaltungsarbeit und Redundanz erzwingt, hat sich die Sprache im Schaltungsentwurf erfolgreich neben VHDL behauptet.

Die wesentlichen Beschreibungselemente von Verilog sind Module, deren Instanzen und parallel arbeitende Anweisungsprozesse. Schalttransistoren und Logikgatter gehören als vordefinierte Primitivmodule ebenso zum Sprachumfang. Verilog-Beschreibungen können somit ein weites Spektrum zwischen verhaltens- und strukturorientierten Entwurfsdarstellungen umfassen.

In der Sprachdefinition sind Datentypen verankert, die Daten mit ein oder mehreren Bit Breite und Felder derartiger Daten verarbeiten. Diese basieren auf einer vierwertigen Darstellung von digitalen Signalen mittels der Zustände 0, 1, hochohmig (Z) und undefiniert (X). Für Verbindungen und verdrahtete Logik können außerdem noch Treiberstärken in vier Abstufungen und Kapazitäten in drei Größen angegeben werden. Für die Darstellung ganzer Zahlen und realer Zahlen sind ebenfalls Datentypen vorhanden. In Kombination mit den Verilog-Primitiven, anderen Modulinstanzen und den Anweisungsprozessen können damit alle Beschreibungsebenen eines Entwurfes von der Algorithmusebene bis hinunter zur Schaltkreisebene abgedeckt werden, letztere aber nur mit eingeschränkter Genauigkeit. Die Nutzung der Datentypen wird durch Synthesesysteme nur im Bereich der realen Zahlen eingeschränkt. Die Definition neuer Datentypen auf der Grundlage der vorgegebenen Typen ist in Verilog nicht möglich.

Verilog besitzt ebenso wie VHDL ein Zeitmodell für Simulationen, welches das Zeitverhalten von Signalen und Anweisungsfolgen darzustellen erlaubt. Durch Sprachelemente zur Beschreibung der Zeit und des reaktiven Verhaltens von Prozessen kann synchrone und asynchrone Logik beschrieben werden. Jedoch werden auch hier Verzögerungen von Synthesesystemen vernachlässigt und die

Unterscheidung von synchroner und asynchroner Logik auf der Grundlage der Prozeßaktivierungen vorgenommen.

Im Vergleich mit VHDL ist Verilog unter Beachtung der Einschränkungen von Synthesesystemen als gleichwertige Beschreibungssprache zu betrachten. Verilog-Beschreibungen sind aufgrund des einfacheren Sprachkonzeptes meist einfacher und schneller zu erstellen als vergleichbare VHDL-Beschreibungen und in der Simulationsgeschwindigkeit überlegen. Dafür bietet VHDL bei größeren Entwurfsprojekten, an denen mehrere Entwickler arbeiten, aufgrund der Sichten, der Bibliotheksverwaltung und der strengen Typenprüfung Vorteile im Austausch, der Dokumentation und der Wiederverwendung von Entwurfsteilen.

In dieser Arbeit wird die Sprache VHDL nicht weiter betrachtet werden. Jede synthesegeeignete Verilog-Beschreibung läßt sich ebensogut in VHDL codieren und umgekehrt, weswegen eine Differenzierung keine zusätzlichen Erkenntnisse für diese Arbeit erwarten läßt.

Protocol-Compiler-HDL: Der Protocol-Compiler ist ein 1997 von Synopsys Inc. eingeführtes Entwurfswerkzeug für Controllerschaltungen zur Verarbeitung von Kommunikationsprotokollen [SeaHol96]. Die Entwurfseingabe erfolgt durch Anreihung bzw. Verschachtelung graphischer Elemente, vergleichbar mit der Aufstellung von Grammatiken für formale Sprachen. Das Aussehen der Entwürfe ist der in gedruckten Protokollspezifikationen benutzten Darstellungsweise sehr ähnlich. Die Protocol-Compiler-HDL ist an das Werkzeug gebunden und der Austausch von Entwürfen mit anderen Programmen erfolgt über Verilog oder VHDL.

Die Protocol-Compiler-HDL baut im wesentlichen auf der Methode der *production based specification* (PBS) auf. Hierbei werden synchrone Automaten auf der Basis von grammatischen Produktionen formaler Sprachen beschrieben [SeaBre94][SeaBre92]. Zu den Grundelementen der Sprache gehören Frames, die mit Symbolen in formalen Sprachen vergleichbar sind und durch Produktionen anderer Frames definiert werden, an die Frames gebundene Aktionen und Operatoren zur Anreihung, Selektion und Wiederholung der Frames. Die Struktur der Frames in Hierarchie oder Produktionsaufbau hat kaum Auswirkungen auf die Struktur des vom Protocol-Compiler in Verilog oder VHDL generierten Controllers, der als hierarchiefreie, monolithische FSM erzeugt wird. Die HDL des Protocol-Compilers erlaubt im wesentlichen nur eine verhaltensorientierte Beschreibung. Durch Einbindung von *user defined actions* oder Instanzen in den Controller, die in Verilog bzw. VHDL vorher außerhalb des Protocol-Compilers entwickelt wurden, kann ein Entwurf mit dem Protocol-Compiler begrenzt strukturorientiert beschrieben werden. Es handelt sich dann aber um keinen reinen Protocol-Compiler-Entwurf mehr.

Die Datentypen des Protocol-Compilers ermöglichen die Darstellung von Bits, und Bitvektoren, die entsprechend der von Verilog bekannten vierwertigen Logik mit 0, 1, undefiniert (X) und hochohmig (Z) belegt werden können. Treiberstärken, Kapazitäten oder selbstdefinierte Datentypen unterstützt der Protocol-Compiler nicht. Er ist somit nicht nur durch die zeitliche Bindung elementarer Frames und

Aktionen an Takte auf die RT-Ebene beschränkt, sondern auch durch seine Datentypen. Die mit dem Protocol-Compiler erstellbaren Entwürfe sind dafür aber mit RTL-Synthesesystemen in jedem Fall problemlos verarbeitbar.

Die zeitliche Bindung elementarer Frames und Aktionen an Takte beschränkt die Protocol-Compiler-HDL auf den Entwurf rein synchroner Logik. Indirekt über Instanzen externer Module in Verilog bzw. VHDL kann asynchrones zeitliches Verhalten in die Entwürfe eingebracht werden. Jedoch wird hierbei der eigentliche Sprachumfang des Protocol-Compilers verlassen.

2.2.2 Simulatoren

Ein Simulator ist im Schaltungsentwurf ein Programmsystem, welches auf der Grundlage der Beschreibung einer Schaltung ihr Verhalten als Reaktion auf die angelegten Eingangswerte und die fortschreitende Zeit berechnet. Die Schaltungsbeschreibung dient in der Simulation als Modell der realen Schaltung und kann in Verhaltensdarstellung, Strukturauflösung und anderen Eigenschaften sämtliche Möglichkeiten der jeweiligen HDL voll ausschöpfen.

Simulatoren und Modelle spielen beim Entwurf integrierter Schaltungen eine zentrale Rolle für die Überprüfung manueller und automatisierter Entwurfsschritte, z.B. nach Synthese und Layout in einer Post-Layout-Verifikation unter Beachtung präziser Laufzeitdaten. Nur so ist trotz der beständig wachsenden Komplexität im Schaltungsentwurf das Ziel erreichbar, einen integrierten Schaltkreis bereits im ersten Anlauf ohne Entwurfsfehler fertigen zu können (*first time right*).

Von den zahlreichen für den Schaltungsentwurf verfügbaren Simulatoren sind hier wegen der bereits vorgenommenen Einschränkung im Sprachbereich jedoch nur noch Simulatoren für die Hardwarebeschreibungssprache Verilog wichtig. Der Protocol-Compiler selbst verfügt über keinen eigenen Simulator, sondern bindet vorhandene Verilog- oder VHDL-Simulatoren in seine Oberfläche ein. Diese Einbindung stellt an die verwendeten Simulatoren besondere Anforderungen. Im Fall von Verilog muß der Simulator das *programming language interface* (PLI) nach IEEE Std 1364-1995 für die Rechnerplattform des Protocol-Compilers vollständig unterstützen. Daher konnte als einheitlicher Simulator für alle in dieser Arbeit betrachteten Entwurfsabläufe nur Verilog-XL von Cadence Design Systems verwendet werden [VerCad97].

2.2.3 Synthesewerkzeuge

Im Schaltungsentwurf dienen Synthesewerkzeuge allgemein zur Verfeinerung von Entwurfsbeschreibungen. Sie bilden die Eigenschaften eines gegebenen Entwurfs durch detailliertere Beschreibungsformen nach und nähern so den Gesamtentwurf an die Fertigungsschnittstelle an. Die Werkzeuge unterteilen sich nach ihrer Spezialisierung, der Form der akzeptierten Beschreibungen und den Freiheiten in der Strukturfestlegung.

Neben universellen Synthesewerkzeugen, die je nach Hierarchieebene der Entwurfselemente für Eingangs- und Zielbeschreibungen beliebige Entwürfe auf

Logik-, RT-, Algorithmus- oder Systemebene verarbeiten bzw. ermöglichen gibt es eine Vielzahl spezialisierter Werkzeuge. Diese unterstützen nur Beschreibungen oder Parametrisierungen eng eingegrenzter Entwurfsmuster und bilden diese auf fest in den Werkzeugen verankerte Komponentenstrukturen ab. Beispiele für derartige Werkzeuge, die als *Modulgeneratoren* oder *Core-Generators* bezeichnet werden und neben einer Verbindungsstruktur meist auch ein Layout bestimmen, sind ALU-, Datenpfad-, Speicher- und PLA-Generatoren. Sie sind in der Regel an bestimmte Zieltechnologien und spezielle, sehr beschränkte Eingabe-HDLs gebunden. In einem synthesebasierten Entwurfsablauf mit Verilog auf der RT- oder der Algorithmus-Ebene kommen solche Generatoren nur bei der gezielten Erweiterung von *Komponentenbibliotheken* zur Anwendung, die von universellen Synthesewerkzeugen beim Mapping verwendet werden. Sie werden im folgenden daher nicht weiter betrachtet.

Je nach Art der zu synthetisierenden Entwurfsbeschreibung hat ein Synthesewerkzeug sehr unterschiedliche Verfahren zur Bestimmung der Zielbeschreibung zu verwenden. In der praktischen Anwendung werden daher für unterschiedliche Beschreibungsformen getrennte Werkzeuge benutzt. Die von einem Werkzeug zu verarbeitenden Beschreibungen sind daher auch dann konsistent in einer Form auszuführen, wenn sich hinter einer Benutzeroberfläche mehrere Werkzeuge verbergen, von denen je nach Form der Eingangsbeschreibung eines ausgewählt wird.

Für die Schaltungssynthese auf RT- und Algorithmus-Ebene haben mehrere Werkzeuge nennenswerte Verbreitung gefunden. Für die derzeit dominierende RTL-Synthese sind dies:

- Design-Compiler von Synopsys
- Leonardo von Exemplar Logic
- FPGA-Express von Synopsys
- Galileo von Exemplar Logic
- Synplify von Synplicity

Hierbei sind nur die ersten beiden Werkzeuge, Design-Compiler und Leonardo, universell für Verilog und VHDL und alle Zieltechnologien geeignet. Die übrigen unterstützen Verilog und VHDL ebensogut, sind aber in der Zieltechnologie auf *Field Programmable Gate Arrays* (FPGAs) und *Complex Programmable Logic Devices* (CPLDs) beschränkt. Neben den genannten Werkzeugen gibt es noch viele weitere, die jedoch in der HDL-Eingabe und den verwendbaren Zieltechnologien sehr spezialisiert sind und deswegen kaum Verbreitung gefunden haben.

Für die Synthese algorithmischer Beschreibungen (High-Level-Synthese) gibt es derzeit nur zwei einigermaßen universelle, verbreitete Werkzeuge:

- Behavior-Compiler von Synopsys
- Monet von Mentor Graphics

Diese Werkzeuge können in Verilog oder VHDL formulierte Anweisungsfolgen von Algorithmusebene auf RTL-Komponenten abbilden. Die Beschreibungen sind

dabei auf sich ständig wiederholende, sequentielle Prozesse, wenige Grundformen von Zeitkontrollen und RTL-Unterkomponenten beschränkt. Andere Synthesewerkzeuge für algorithmische Beschreibungen sind in den Anweisungen, dem Anwendungsgebiet oder der Einbindung in vollständige Entwurfsabläufe sehr spezialisiert und werden daher nicht berücksichtigt.

Für die Syntheseuntersuchungen dieser Arbeit wurden der Design-Compiler und der Behavior-Compiler ausgewählt. Gründe hierfür waren die Verfügbarkeit, die bereits vielfach erfolgreiche Anwendung in vollständigen Entwurfsabläufen und das reibungslose Zusammenspiel mit anderen Werkzeugen, wie dem Protocol-Compiler oder den FPGA-Layoutwerkzeugen von Xilinx. Im folgenden wird die RTL-Synthese und High-Level-Synthese primär im Zusammenhang mit diesen beiden Werkzeugen betrachtet.

2.2.4 Einbettung der Synthese in Entwurfsabläufe

Eine wichtige Voraussetzung für die erfolgreiche Durchführung eines Entwurfs mit Synthese ist das korrekte Zusammenspiel von Sprachen und Werkzeugen in der Abfolge ihrer jeweiligen Anwendung, dem sogenannten Entwurfsablauf oder auch *design flow*. Die in dieser Arbeit angewendeten Abläufe für Controller-, High-Level- und RTL-Synthese sollen hinsichtlich der Werkzeuge, ihres Zusammenspiels und der Entwurfsnotationen nun genauer vorgestellt werden.

Als Zieltechnologien wurden die Gate-Array-Familie LSI10K von LSI Logic und FPGAs von Xilinx benutzt. Die LSI10K-Technologie ist aufgrund ihres mittlerweile zehnjährigen Alters zwar technisch überholt und wird nicht mehr gefertigt, ist jedoch als Bewertungsmaßstab für Syntheseergebnisse aus mehreren Gründen immer noch geeignet. So gehören ihre Komponentenbibliotheken zum Lieferumfang des Design-Compilers und sind problemlos verfügbar. Des weiteren sind die LSI10K-Bibliotheken in ihrem Komponentenumfang und dem Größenmaß auf Gatterbasis aktuellen Bibliotheken für maskenprogrammierte Gate-Arrays sehr ähnlich. Außerdem sind Laufzeitabschätzungen wegen der hohen Gatterlaufzeiten im Vergleich zu den Verdrahtungsverzögerungen ohne Erstellung eines Layouts mit ausreichender Genauigkeit möglich, anders als bei den modernen Submikrontechnologien [Shepar97]. Dies ist insbesondere wegen des komplexen Layoutvorgangs von maskenprogrammierbaren Gate-Arrays günstig, der sich sehr zeitraubend gestalten kann und für größere Testreihen von Entwürfen nicht in Frage kommt. Die Laufzeitabschätzung und Größenbestimmung für LSI10K-Entwürfe wurde im Design-Compiler vorgenommen, der in den entsprechenden Entwurfsabläufen das abschließende Werkzeug war.

Zur Berücksichtigung von moderneren Zieltechnologien und Layouteffekten in den Laufzeiten wurden FPGAs der Serien XC3000 und XC4000 von Xilinx [Xilinx96] verwendet. Für diese stand neben aktuellen Zielbibliotheken die Software zur Verfügung, um eine Schaltung vollständig auf einem FPGA zu plazieren und zu verdrahten, wodurch der Verdrahtungsaufwand und die Laufzeiten genau bestimmt werden konnten. Weiterhin wurde die Durchführung von Post-Layout-Simulationen durch Bibliotheken und Software vollständig unterstützt, so daß die Korrektheit der Syntheseergebnisse nachprüfbar war. Die

zügige, automatisierte Ausführung des gesamten Entwurfsablaufs einschließlich Plazierung, Verdrahtung und Timinganalyse erlaubte außerdem die Betrachtung umfangreicher Versuchsreihen. Damit ergab sich als Abschluß der durchgeführten Entwurfsabläufe für Xilinx-FPGAs die Layoutphase in Bild 2.5.

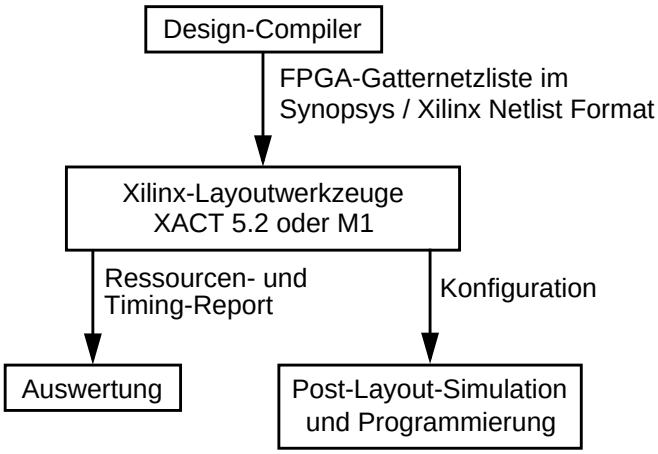


Bild 2.5: Layoutphase in Entwurfsabläufen für Xilinx-FPGAs

Innerhalb der jeweils verwendeten Xilinx-Layoutwerkzeuge wird die vom Design-Compiler erzeugte Gatterstruktur zunächst auf Korrektheit für das verwendete FPGA geprüft. Danach werden die Gatter auf die *combinatorial logic blocks* (CLBs) des FPGAs verteilt, welche durch eine Plazierung und Verdrahtung in die CLB-Matrix des Chips eingeordnet werden. Abschließend generieren die Werkzeuge die Konfiguration für den FPGA-Chip sowie Berichte über den Verbrauch an Logikressourcen und die aus der Plazierung und Verdrahtung resultierenden Laufzeiten. Die Konfiguration wird zur Bestimmung einer Gatternetzliste für die Post-Layout-Simulation und zur Programmierung von FPGA-Chips benutzt.

Der Design-Compiler ist in allen verwendeten Entwurfsabläufen ein zentrales Werkzeug. Er führt neben der Synthese von Verilog-RTL-Beschreibungen auch das Mapping in eine Gatternetzliste auf Logikebene und deren Optimierung für die Abläufe der RTL-, der High-Level-, und der Controllersynthese aus.

Die RTL-Synthese ist in den einfachsten dieser Entwurfsabläufe eingebettet, der sich nur gering von der bereits vorgestellten Grundform unterscheidet (Bild 2.6).

Bei diesem Entwurfsablauf wird das Verilog-Modell eingebunden in eine Testumgebung simuliert, welche die geplante Beschaltung sowie Beobachtungs- und Verifikationshilfen bereitstellt. Die Stimuli der Simulation können fest in dieser Umgebung verankert sein oder durch Parameterdateien geliefert werden. Die Umgebung ist ebenfalls in Verilog beschrieben, jedoch in beliebiger vom Simulator akzeptierter Form, ohne Bindung an die Register-Transfer-Ebene oder die Begrenzungen des Synthesewerkzeugs. Der übrige Ablauf entspricht der bereits beschriebenen Vorgehensweise.

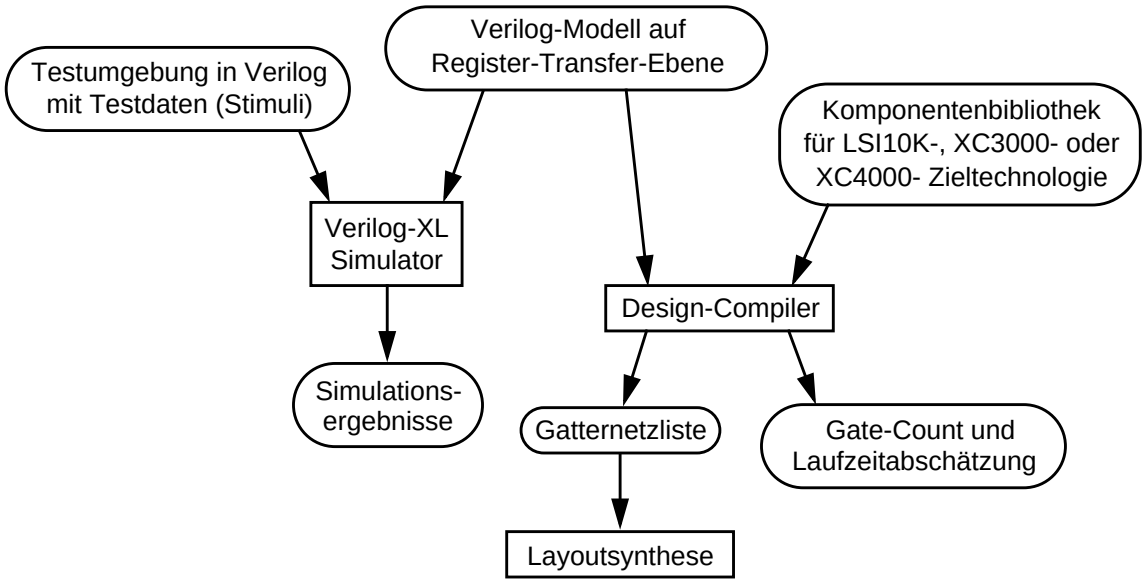


Bild 2.6: Entwurfsablauf der RTL-Synthese mit dem Design-Compiler

Die High-Level-Synthese von Beschreibungen der Algorithmusebene erweitert diesen Entwurfsablauf um den Behavior-Compiler und erhöht den Ansatzpunkt der Beschreibungsform, der Zeit- und der Verhaltensdarstellung (Bild 2.7). Der übrige Ablauf von der Verwendung des Design-Compilers an bleibt gleich.

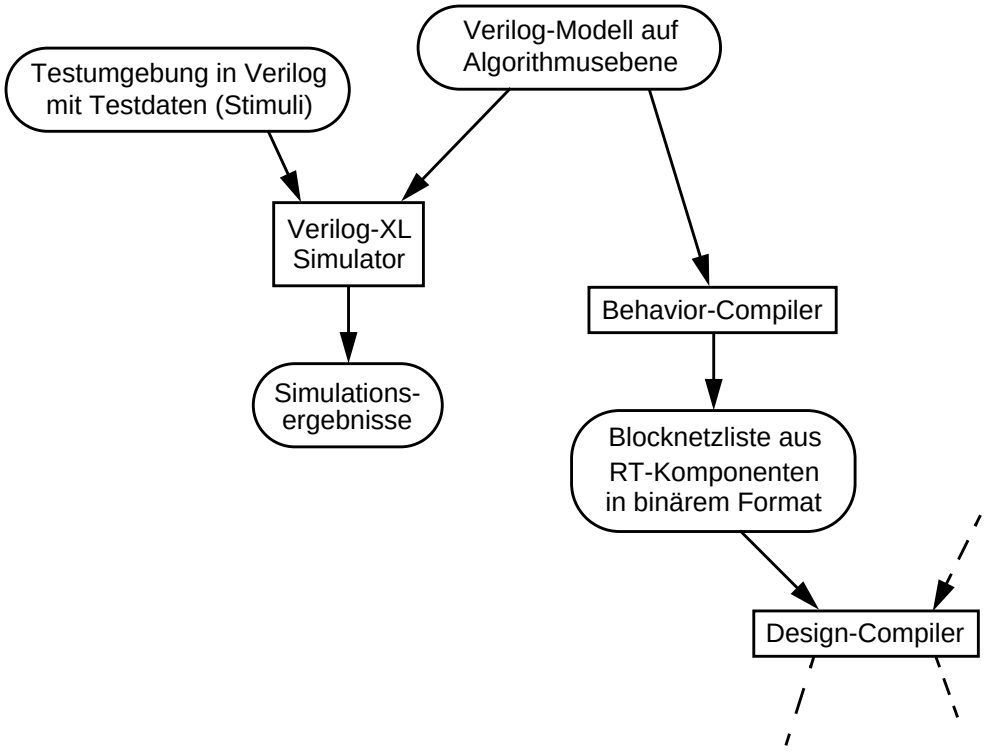


Bild 2.7: Entwurfsablauf der High-Level-Synthese mit dem Behavior-Compiler

Der Austausch der vom Behavior-Compiler generierten RTL-Beschreibung mit dem Design-Compiler erfolgt in einem binären Format für Entwurfsdaten von Synopsys. Der Design-Compiler überführt diese Netzliste aus RT-Komponenten im Anschluß mit Logiksynthese und Technology-Mapping in eine Gatternetzliste auf der Logikebene.

Bei der Controllersynthese mit dem Protocol-Compiler ist keine derart klare Trennung zwischen Daten und Werkzeugen vorhanden, da die Eingabe-HDL an das Werkzeug gebunden ist und die Simulation in die Entwurfsoberfläche integriert wurde (Bild 2.8). Der von der RTL-Synthese her bekannte Ablauf wird auch hier um ein zusätzliches Werkzeug bereichert. Neben der Beschreibungsform ändern sich durch den Verlust der Logikebene für Entwurfselemente die Möglichkeiten zur Darstellung der Zeit und zur Auflösung der Struktur.

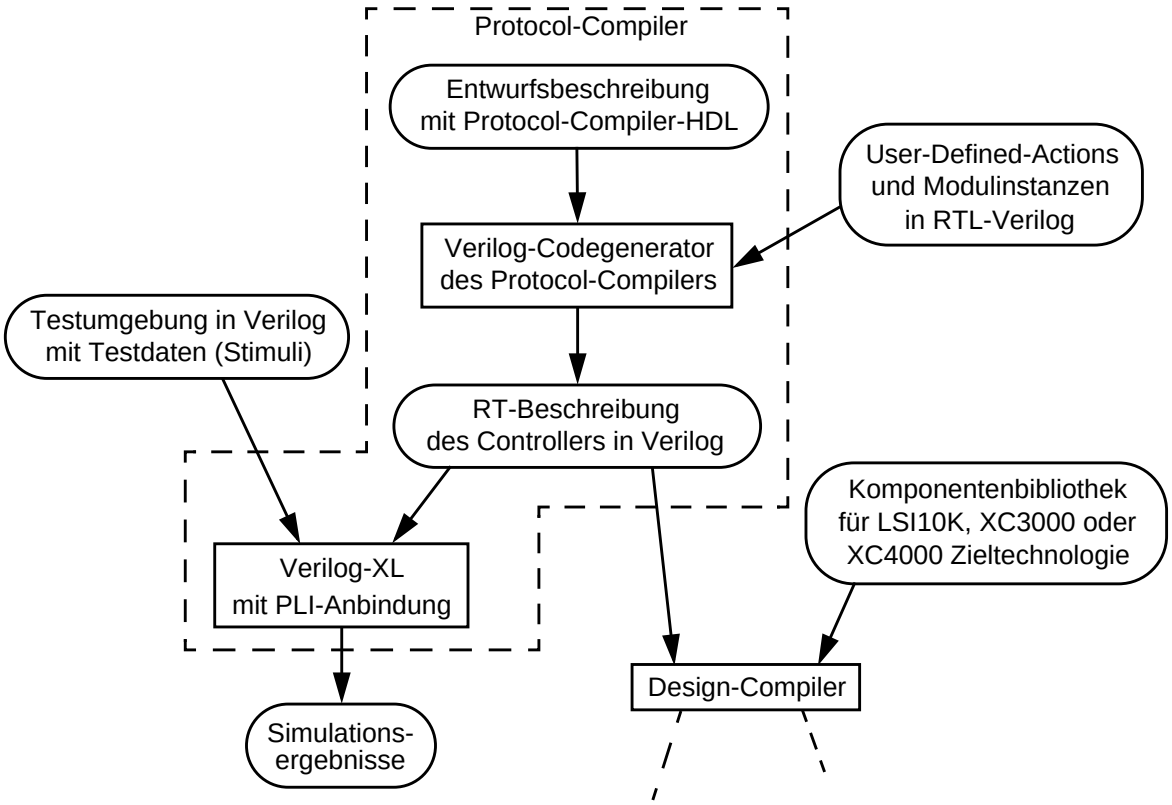


Bild 2.8: Entwurfsablauf der Controllersynthese mit dem Protocol-Compiler

Die Eingabe des Entwurfs, seine Übersetzung in eine optimierte Beschreibung in RTL-Verilog und seine Simulation erfolgen innerhalb der graphischen Umgebung des Protocol-Compilers. Die Erstellung der Testumgebung, der Testdaten und der in den Controller eingebundenen Verilog-Modelle wird wie bei den anderen Entwurfsabläufen getrennt von den verarbeitenden Werkzeugen vorgenommen. Bei der Controllersynthese wird das Controllermodell durch den Codegenerator des Protocol-Compilers in ein RTL-Verilog-Modell überführt, das im Anschluß mit dem Design-Compiler auf eine Gatternetzliste abgebildet wird.

2.3 Die Sprachkonzepte von Verilog und der Protocol-Compiler-HDL

Bisher wurden die in dieser Arbeit verwendeten HDLs nur in einigen wenigen, charakteristischen Eigenschaften umrissen. Die Sprachelemente von Verilog und der Protocol-Compiler-HDL sowie ihre Verwendung zur Modellierung von Entwurfseigenschaften werden in diesem Abschnitt einführend vorgestellt. Dabei wird zunächst auf das Sprachkonzept von Verilog unter Berücksichtigung der bei der Synthese zu beachtenden Einschränkungen eingegangen. Die Protocol-Compiler-HDL wird nachfolgend im Vergleich betrachtet.

2.3.1 Das sprachliche Grundkonzept von Verilog

Verilog ist eine textuelle HDL, deren wesentliches Mittel zur Organisation von Entwurfsteilen und ihrer hierarchischen Zusammensetzung *Module* sind. Jegliche Beschreibung von Entwurfseigenschaften erfolgt innerhalb von Modulgrenzen, die durch die Schlüsselworte `module` und `endmodule` gesetzt sind. Die Module sind abgeschlossene, parallel zueinander arbeitende Entwurfsteile, die mit ihrer Umgebung über eine im Modulkopf festgelegte Schnittstelle kommunizieren, den *Ports*. Über die Ports können in Modulen instanzierte Module Verdrahtungen gleichend sowohl miteinander als auch mit Registern und Anweisungen verbunden werden, die ebenfalls Teil eines Moduls sein können. Ports besitzen eine fest definierte Bitbreite und können als Eingang (`input`), Ausgang (`output`) oder bidirektional (`inout`) angelegt werden.

Die Datentypen von Verilog unterteilen sich in *Netze* zur Verbindung von Komponenten (`wire` und `tri`) sowie in *Register* zur Speicherung von Werten (`reg` und `integer`). Der Registertyp `integer` besitzt eine festgelegte Breite von 32 Bit, während die anderen Datentypen auf ein Bit voreingestellt sind, aber beliebig breiter deklariert werden können. Alle diese Datentypen unterstützen eine vierwertige Logikdarstellung für Bits mit den Werten 0, 1, undefiniert (X) und hochohmig (Z). Netze können im Gegensatz zu Registern keine Daten speichern, sondern reichen lediglich Daten weiter, repräsentieren also reine Verdrahtung. Ihre jeweiligen Werte bestimmen sich aus Portanbindungen oder kontinuierlichen Zuweisungen kombinatorischer Berechnungsergebnisse. Die Register können je nach zeitlichem Verhalten ihrer Wertzuweisungen als Zwischenvariablen bei kombinatorischen Berechnungen oder als Modell für Hardwarespeicher dienen.

Ablaufbeschreibungen erfolgen in Blöcken prozeduraler Anweisungen für Zuweisung, Selektion und Wiederholung. Diese Blöcke können als Anweisungsprozesse parallel zueinander arbeiten oder den Rumpf einer `task` bzw. `function` bilden. Letztere werden von Prozessen oder zeitkontinuierlichen Zuweisungen (*continuous assignments*) aufgerufen. Blöcke, die durch das Schlüsselwort `always` als Anweisungsprozesse deklariert wurden, entsprechen im Bezug auf die untereinander parallele und permanent wiederholte Abarbeitung den Eigenschaften von Schaltungselementen. Durch Synchronisation der Anweisungsprozesse mit Ereignissen wie Wertänderungen oder Signalflanken kann das reaktive Verhalten von Hardware sehr präzise beschrieben werden. Auf RT-Ebene ist hierdurch die Unterscheidung zwischen kombinatorischer Logik und Speicherelementen durch

die Art der Prozeßsynchronisation möglich. Die Synchronisation von Anweisungen mit Ereignissen erfolgt mit der Verilog-Anweisung @, der als Parameterliste die auf Ereignisse zu überwachenden Netze und Register nachgestellt sind. Diese Liste wird im folgenden als *Sensitivitätsliste* bezeichnet. Die @-Anweisung kann dabei entweder nach dem Schlüsselwort `always` plaziert werden (Design-Compiler) oder innerhalb des Prozesses vor Anweisungen (Behavior-Compiler).

Zusätzlich zum algorithmischen und reaktiven Verhalten kann mit Verilog auch zeitliches Verhalten beschrieben werden. Die Warte-Anweisung # erlaubt die gezielte Verzögerung einer Anweisung um die angegebene Zahl von Simulationszeiteinheiten. Im Unterschied zur Prozeßsynchronisation wird der Zeitpunkt der Fortsetzung nicht durch ein Ereignis außerhalb des Prozesses bestimmt, sondern durch die fortschreitende Simulationszeit. Verzögerungen werden sowohl bei der Erzeugung von Stimuli in Testmodulen als auch zur Nachbildung von reaktiven Verzögerungen in Anweisungsprozessen eingesetzt, jedoch bei der Synthese gänzlich ignoriert.

Neben den Sprachelementen zur ablauforientierten Modellierung verfügt Verilog über vordefinierte, instanzierbare Module für logische Verknüpfungen. Dies erlaubt die Konstruktion von strukturorientierten Beschreibungen, die lediglich aus Instanzen und deren Verdrahtung durch Netze bestehen und als Netzlisten bezeichnet werden.

Die Protokollierung von Daten auf textuelle oder graphische Anzeigen und der Zugriff auf Dateien des als Simulationsbasis verwendeten Rechnersystems erfolgt über vordefinierte Systemtasks und -funktionen. Diese sind an ihrer Einleitung durch das $-Zeichen erkennbar und dienen zur Erstellung von Testumgebungen für Hardwarebeschreibungen. Sie werden von Synthesewerkzeugen überlesen.

Bei ausschließlicher Verwendung dieser synthesefähigen Sprachelemente kann es dennoch zu Abweichungen zwischen dem Verhalten eines Verilog-Modells und der synthetisierten Schaltung kommen [MilCum99]. Wird beispielsweise ein Register aus parallel aktiven Anweisungsblöcken konkurrierend beschrieben, so führt dies zu einer vom Simulator abhängigen, eventuell nichtdeterministischen Abfolge zweier Zuweisungsoperationen. Der Design-Compiler konstruiert hierfür eine Verbindung der Ausgänge parallel aktiver Register, die nur für bestimmte Wertbelegungen und Technologien mit Wired-Logic-Fähigkeit dem Verhalten des Modells entspricht.

Die Synchronisation von Prozessen und Anweisungen mit der Warte-Anweisung # kann ebenso zu unerwarteten Syntheseergebnissen führen, da die hierdurch für den Simulator modellierten Abhängigkeiten der Synthese gänzlich verborgen bleiben. Neben verdrahteter Logik aufgrund konkurrierender Zugriffe kann dies auch in der Auslassung von Registerzuweisungen resultieren, deren Ergebnisse aus Synthesesicht ohne eine Verzögerung durch andere Werte überschrieben werden.

Auch unvollständige Sensitivitätslisten kombinatorischer Logiken, die nicht jedes gelesene Signal enthalten, führen zu Inkonsistenzen zwischen Pre- und Post-Synthese-Simulationen. Die Synthese erzeugt in diesem Fall ein kombinatorisches

Netz auf der Grundlage der Datenabhängigkeiten, während für die Simulation ein asynchrones Speicherelement entsteht, welches durch alle Wertänderungen in der Sensitivitätsliste getaktet wird. Ebenso führen auch Zuweisungen des Wertes X oder von Systemtasks zu Abweichungen, da diese Zuweisungen gar nicht oder mit X als Platzhalter für beliebige Werte (*don't care*) umgesetzt werden.

Trotz der Synthesefähigkeit sind derartige Modellierungen aufgrund ihres unterschiedlichen Pre- und Post-Synthese-Verhaltens als fehlerhaft anzusehen und zu vermeiden. Sie werden in dieser Arbeit daher nicht benutzt und nicht weiter berücksichtigt.

2.3.2 Vergleichende Betrachtung der Protocol-Compiler-HDL

Die Protocol-Compiler-HDL baut auf einer überschaubaren Menge graphischer und textueller Basiselemente auf, die teils stark an die aus formalen Sprachen bekannten Elemente erinnern. Damit die Entwürfe nicht nur auf Stimulidaten reagieren können sondern auch Daten manipulieren können, gibt es zusätzlich Operationen auf Variablen und Ports.

Die Entwurfselemente lassen sich in die drei Klassen *Frames*, *Actions* und *Operatoren* unterteilen. Die sogenannten Frames sind mit den Symbolen in formalen Sprachen vergleichbar. Es gibt primitive Frames, welche die Funktion terminaler Symbole haben, und Referenz-Frames, die den nichtterminalen Symbolen entsprechen. Frames können durch Operatoren zu Sequenzen oder Alternativen zusammengestellt werden, an Bedingungen geknüpft werden oder wiederholt werden.

Die Frames werden durch Rechtecke dargestellt, in denen die Bedingung für die Bearbeitung eingetragen ist (primitive Frames) oder eine Referenz auf eine Frame-Definition (Referenz-Frames). Wird ein primitives Frame bearbeitet, so wird hierdurch ein Takt des zugrundeliegenden Automatenmodells verbraucht. Außerdem kann die Bearbeitung eines Frames mit Aktionen verbunden werden, wie der Berechnung und Zuweisung eines arithmetischen Ausdrucks. Bild 2.9 zeigt einige Beispiele für einfache Frame-Definitionen.

Mit den drei Operatoren „Sequenz“ (geschweiftes Klammerpaar), „Alternative“ (vertikale Balken) und „Wiederholung“ (eckige Klammern mit Exponent) lassen sich in Kombination mit der hierarchischen Struktur relativ leicht komplexere Modelle erstellen. Durch Abwandlung des Wiederholungsexponenten kann die Ausführung optional (kein Exponent) oder beliebig oft erfolgen (∗). Als Aktionen können zudem auch in RTL-Verilog beschriebene Funktionen ausgeführt werden.

Wird die Bedingung eines Frames jedoch nicht erfüllt, so wird dieses Frame nicht bearbeitet und eine es einbettende Frame-Sequenz abgebrochen. Dies ist analog zu der Verwerfung von grammatischen Produktionen bei der Bildung von Ausdrücken in formalen Sprachen.

Neben diesen grundlegenden Entwurfselementen stehen noch zwei weitere leistungsfähige Operatoren zur Verfügung, die bei der Datensynchronisation und Fehlerbehandlung von Kommunikationsprotokollen die Modellierung verein-

fachen. Hierbei handelt es sich zum einen um den Bedingungsoperator (Qualifier), der die Ausführung der von ihm umgebenen Frames nur zuläßt, wenn eine bestimmte Bedingung erfüllt ist. Ansonsten wird die Ausführung abgebrochen. Zum anderen kann mit dem Warte-Operator (RunIdle) die Abarbeitung von Frame-Sequenzen solange angehalten werden, bis eine bestimmte Bedingung erfüllt wird.

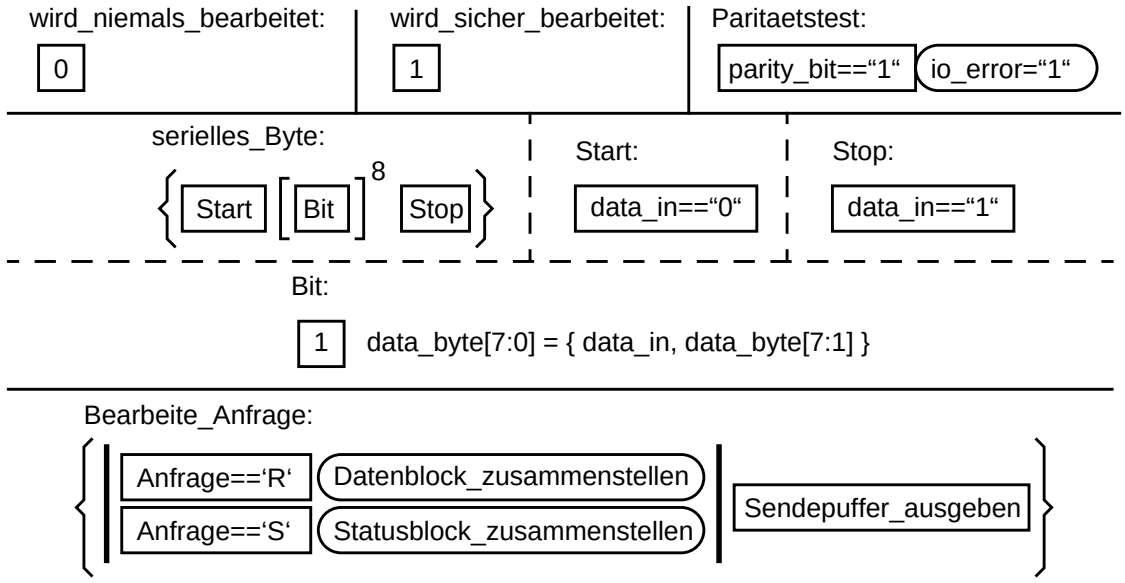


Bild 2.9: Beispiele für einfache Frame-Definitionen

Im Gegensatz zu Verilog gibt es keine Möglichkeit zur Konstruktion mehrerer, parallel arbeitender Entwurfshierarchien, da nur genau ein Frame als Wurzel der Entwurfshierarchie angegeben werden kann. Bei Verilog gilt jedes Modul, das nicht von einem anderen Modul instanziert wird, automatisch als die Wurzel eines Hierarchiebaumes.

Das Timing des gesamten Entwurfes ist synchron an einen Takt gebunden, wobei die Ausführung jedes primitiven Frames genau einen Taktzyklus benötigt. Trotz gleichzeitiger Ausführung mehrerer Frames und Actions in einem Takt gibt es keine wirkliche Nebenläufigkeit in der Ausführung. Stattdessen werden sowohl alternative Frames als auch gleichzeitige Aktionen sequentiell entsprechend ihrer Anordnung in den Frame-Definitionen abgearbeitet. Damit ist zwischen Frames und zwischen Aktionen eine Priorisierung vorhanden, die Nichtdeterminismen, wie z.B. bei konkurrierenden Schreibzugriffen auf eine Variable, ausschließt.

Infolge des streng synchronen Zeitschemas, den Prioritäten zwischen Frames und Actions und der auf einem Mealy-Automaten basierenden Realisierung sind Unterschiede im Verhalten zwischen Simulation und Synthese bzw. nach den Transformationen in eine Verilog-Beschreibung oder eine Gatternetzliste nahezu ausgeschlossen. Die Verwendung nicht initialisierter Variablenwerte in Berechnungen ist die einzige Möglichkeit, solche Unterschiede hervorzurufen. Wegen des damit insgesamt undefinierten Verhaltens des Automaten ist dies jedoch als Beschreibungsfehler zu betrachten.

2.4 Quantifizierung und Bewertung von Syntheseergebnissen

Die objektive Bewertung einer Sache erfordert die Existenz und die Anwendung von Maßen für deren qualitätsbestimmende Eigenschaften, so auch bei Syntheseergebnissen. In der integrierten Halbleiterschaltungtechnik werden üblicherweise Eigenschaften wie die Fläche einer Schaltung, die Anzahl ihrer Logikelemente, die Schaltgeschwindigkeit und der Energieverbrauch verwendet.

Als gleichsam kombiniertes Maß von Fläche und Anzahl der Elemente kommt dabei besonders oft die Anzahl von NAND-Gatteräquivalenten zur Anwendung, die als *Gate-Count* bezeichnet wird.

Die Schaltgeschwindigkeit wird je nach Schaltungstyp über die Verzögerung von den Eingängen zu den Ausgängen einer Schaltung oder deren Taktungsrate angegeben.

Die beiden vorgenannten Maße sind im synthesebasierten Schaltungsentwurf besonders oft anzutreffen und in Synthesewerkzeugen oder Layoutwerkzeugen mit wenigen Arbeitsschritten zu gewinnen. Sie wurden deshalb in dieser Arbeit als hauptsächliches Ordnungskriterium im Lösungsraum verwendet.

Zur Bestimmung der beiden Maße in den Experimenten dieser Arbeit wurden als Zieltechnologien neben der LSI10K-Technologie für maskenprogrammierte ASICs die Xilinx-FPGAs der Serien XC3000 und XC4000 benutzt. Während bei der LSI10K-Technologie die Schaltungsgröße mit dem Gate-Count angegeben wird, werden bei den FPGAs die Logikblöcke gezählt, die im Vergleich zu NAND-Gattern eine um das zehnfache gröbere Granularität besitzen. Außerdem sind aufgrund der unterschiedlichen Nutzungsmöglichkeiten von FPGA-Logikblöcken (nur Kombinatorik, nur Register, Register und Kombinatorik oder Verdrahtungsersatz) Aussagen über die tatsächlich benötigte Logik immer mit gewissen Fehlern behaftet. Es ergab sich zwangsläufig die Frage, inwieweit das FPGA-Größenmaß für den Logikverbrauch in *combinatorial logic blocks* (CLBs) mit dem weithin akzeptierten Gate-Count vergleichbar ist.

Zur Beantwortung dieser Frage wurden verschiedene Schaltungsentwürfe mit dem Design-Compiler auf die drei Zieltechnologien LSI10K, XC3000 und XC4000 abgebildet und die resultierenden Größenwerte statistisch auf Korrelation geprüft.

2.4.1 Meßreihen zum Vergleich von Gate-Count und CLB-Count

Mehrere Schaltungsentwürfe, die als synthesefähige RTL-Verilog-Beschreibung vorlagen und in den letzten Jahren an der Abteilung E.I.S. im Rahmen von Forschung und Lehre entstanden sind, bildeten die Grundlage für den Vergleich von Gate-Count und CLB-Count. Die jeweils höchsten zwei Hierarchieebenen der Entwürfe wurden mit dem Design-Compiler hinsichtlich folgender Kombinationen von Zieltechnologien und Maßen untersucht:

- LSI10K-Bibliothek mit Gatterzählung durch den Design-Compiler
- Xilinx-XC4000 mit Logikzellenzählung durch den Synopsys FPGA-Compiler
- Xilinx-XC4000 mit CLB-Zählung durch Xilinx-XACT
- Xilinx-XC3000 mit Logikzellenzählung durch den Synopsys FPGA-Compiler

- Xilinx-XC3000 mit CLB-Zählung durch Xilinx-XACT

Es wurden keine Makrozellenbibliotheken wie Xilinx-XBLOX für XC4000-FPGAs eingesetzt, da entsprechende Bibliotheken für die Technologien LSI10K und XC3000 nicht verfügbar waren. Die Syntheseläufe wurden ohne Structuring und Flattening und mit minimaler Flächenforderung durchgeführt. Es wurden keine Betriebsbedingungen oder Verdrahtungsmodelle gesetzt. Der Mapping-Effort der Synthese wurde hoch gewählt. Die resultierenden Ergebnisse für Gesamtzahl, kombinatorische Logik (CL), Flip-Flops (FF) und FPGA-CLBs (CLB) zeigt Tabelle 2.1.

Entwurf	LSI10K Gatter; CL + FF	XC4000 Zellen; CL + FF	XC4000 CLB; FF	XC3000 Zellen; CL + FF	XC3000 CLB; FF
CRTC	796; 354+442	96.5; 73.5+23	72; 46	96.5; 73.5+23	70; 46
-[hcount]	280; 144+136	32.5; 26+6.5	25; 13	32.5; 26+6.5	25; 13
-[memacc]	404; 142+262	35; 24+11	25; 22	35; 24+11	21; 22
-[vcount]	273; 130+143	27.5; 22+5.5	20; 11	27.5; 22+5.5	22; 11
DAR	1227; 264+963	132.5; 79+53.5	100; 107	133.5; 80+53.5	106; 107
-[BIPDEC]	389; 91+298	34.5; 19.5+15	30; 30	34.5; 19.5+15	36; 30
-[CONTROL]	243; 79+164	23.5; 17.5+6	16; 12	23.5; 17.5+6	24; 12
-SHIFTIN]	367; 37+330	31; 16+15	20; 30	31; 16+15	16; 30
-[DACOUT]	413; 130+283	38.5; 26+12.5	30; 25	38.5; 26+12.5	27; 25
DCF-77	2746; 1212+1534	343; 285.5+57.5	196; 115	343.5; 286+57.5	283; 115
-[bit_rec]	321; 112+209	27; 19+8	16; 16	27; 19+8	16; 16
-[frm_rec]	451; 139+312	34; 22+12	25; 24	34; 22+12	26; 24
-[seg_drv]	44; 44+0	11.5; 11.5+0	6; 0	11.5; 11.5+0	13; 0
-[lcd_drv]	1213; 481+732	104.5; 91.5+13	85; 26	104.5; 91.5+13	118; 26
-[clk_100]	269; 119+150	27.5; 20+7.5	20; 15	27.5; 20+7.5	17; 15
-[tim_cnt]	814; 377+437	97; 80+17	81; 34	97; 80+17	99; 34
URISC	2182; 1122+1060	286.5; 250.5+36	269; 72	286; 250+36	263; 72
-[ALU]	924; 690+234	142.5; 133.5+9	169; 18	142.5; 133.5+9	136; 18
-[IEU]	1161; 470+691	111.5; 84.5+27	100; 54	111.5; 84.5+27	111; 54
-[BL]	129; 129+0	8; 8+0	16; 0	8; 8+0	16; 0
MRISC	2675; 1559+1116	357; 305+52	324; 104	354; 302+52	349; 104
-[ALU]	1096; 862+234	174.5; 165.5+9	196; 18	174.5; 165.5+9	168; 18
-[IEU]	1170; 479+691	112; 85+27	100; 54	112; 85+27	113; 54
-[BL]	674; 194+480	60; 44+16	62; 32	60; 44+16	84; 32
TETCON	2660; 806+1854	253; 163.5+89.5	196; 174	252.5; 163+89.5	200; 174
-[BUSCON]	637; 48+589	43; 21+22	41; 39	43; 21+22	37; 39
-[CNTDWN]	457; 193+264	45; 33+12	36; 24	45; 33+12	31; 24
-[KEYCON]	722; 276+446	62; 41.5+20.5	56; 41	62; 41.5+20.5	58; 41
-[LCDCON]	951; 267+684	79; 52+27	64; 54	79; 52+27	58; 54
-[SUPCON]	299; 91+208	19; 11+8	20; 16	19; 11+8	18; 16

Tabelle 2.1: Ergebnisse der Zielbibliotheksvariation

2.4.2 Korrelation von Gate-Count und CLB-Count

Durch die Berechnung der einfachen, linearen Korrelation zwischen den einzelnen Werten wurde untersucht, wie verläßlich die FPGA-Maßangaben in statistischer Hinsicht mit der Gatteranzahl zusammenhängen.

Der Korrelationskoeffizient r ergibt sich nach [Renner81] für zwei n-elementige Reihen $x = (x_1, x_2, \ldots, x_n)$ und $y = (y_1, y_2, \ldots, y_n)$ zu

$$r = \frac{n \cdot \sum(x \cdot y) - \sum x \cdot \sum y}{\sqrt{(n\sum x^2 - (\sum x)^2) \cdot (n\sum y^2 - (\sum y)^2)}}$$

mit

$$\sum x = x_1 + x_2 + \ldots + x_n \qquad \sum y = y_1 + y_2 + \ldots + y_n$$

$$\sum x^2 = x_1^2 + x_2^2 + \ldots + x_n^2 \qquad \sum y^2 = y_1^2 + y_2^2 + \ldots + y_n^2$$

$$\sum(x \cdot y) = x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_n \cdot y_n$$

Tabelle 2.2 zeigt die Korrelationskoeffizienten zwischen Gattern, Zellen und FPGA-Logikblöcken (CLBs).

	XC4000 Zellen	XC4000 CLB; FF	XC3000 Zellen	XC3000 CLB; FF
LSI10K Gatter CL+FF; CL; FF	0.9714; 0.9896; 0.9624	0.9063; ; 0.9625	0.9716; 0.9890; 0.9624	0.9499; ; 0.9625
XC4000 Zellen CL+FF; CL; FF		0.9594; ; 0.9996	0.9999; 0.9999; 1	0.9876; ; 0.9996
XC4000 CLBs			0.9587	0.9744
XC3000 Zellen		0.9871; 0.9996		0.9871; 0.9996

Tabelle 2.2: Korrelation zwischen Gattern, Zellen und CLBs

Nach [Renner81] sind diese Korrelationswerte hoch signifikant, da sie alle dichter am Wert 1 liegen, als der statistische Grenzwert für hohe Signifikanz bei dieser Probenanzahl. Es besteht also ein gesicherter statistischer Zusammenhang zwischen den jeweils in Vergleich gebrachten Maßwerten. Dies ist sowohl bei gemischter als auch separater Betrachtung von Kombinatorik und Registern der Fall. Daher lassen sich die betrachteten FPGA-Maße als Bewertungsalternative zu dem klassischen Gattermaß verwenden, ohne daß dies zu qualitativen Fehlern führt.

3 Register-Transfer-Synthese

In diesem Kapitel wird die Synthese von Register-Transfer-Beschreibungen der Sprache Verilog betrachtet, wobei mit den Sprachkonstrukten für RT-Elemente und ihren Kombinationsmöglichkeiten begonnen wird. Die RTL-Verilog-Konstrukte und die damit beschriebenen Elemente entsprechen der Eingabeform des *HDL Compilers for Verilog* [SynHDL97], der die Eingangsanalyse des Design-Compilers von Synopsys [SynDCR97] für Verilog-Modelle vornimmt. Sie sind ebenso für viele andere Synthesewerkzeuge mit Eingangsbeschreibungen in RTL-Verilog gültig und orientieren sich an dem Zusatz IEEE P1364.1-1999 [VerSyn99] zur Verilog-Norm, dem *Standard for Verilog Register Transfer Level Synthesis*.

Zusammen mit den RTL-Verilog-Konstrukten werden auch die mit ihnen verknüpften Eigenschaften für die RTL-Synthese vorgestellt, wie die Darstellung von zeitlichen Aspekten oder die Auflösung in Hardwarestrukturen. Im Anschluß wird auf die Kombinationsmöglichkeiten dieser Grundkonstrukte und die dabei zu beachtenden Einschränkungen eingegangen, wobei die Problematik abweichender Simulationsergebnisse nach der Synthese besondere Beachtung findet.

Ein weiterer Schwerpunkt dieses Kapitels liegt auf den Optimierungsmöglichkeiten der RTL-Synthese mit dem Design-Compiler und anderen, gängigen RTL-Synthesewerkzeugen. Die Auswirkungen der RTL-Optimierungen wurden in den Syntheseexperimenten dieser Arbeit mit untersucht und bilden als Ergebnisse die Grundlage für spätere Kapitel, auf die hier vorbereitet wird.

3.1 Verilog-Konstrukte für Register-Transfer-Elemente

In diesem Abschnitt wird auf die im vorigen Kapitel eingeführte HDL Verilog hinsichtlich ihrer Konstrukte für RTL-Beschreibungen näher eingegangen. Zu den grundlegenden Möglichkeiten, das Innenleben eines Verilog-Moduls für die RTL-Synthese zu gestalten, gehören:

- Instanzen von Logikprimitiven und selbstdefinierten Modulen
- kontinuierliche Zuweisungen (Continuous-Assignments)
- `always`-Anweisungsblöcke

Außerdem können Anweisungsfolgen über Tasks oder Funktionen aus `always`-Blöcken bzw. kontinuierlichen Zuweisungen ausgelagert werden, sind aber anders als diese keine dauerhaft aktiven Anweisungsprozesse. Sie sind nur für die Zeit ihres Aufrufes aktiv und dürfen keine Zeitkontrollen beinhalten. Ähnlich den Modulen erlauben Tasks und Funktionen die Kapselung von Entwurfsteilen in Blockstrukturen und deren hierarchische Zusammensetzung. Sie bringen im Gegensatz zu Modulen oder Logikprimitiven jedoch keine aktiven Ausführungsprozesse in den Entwurf ein, die zur Beschreibung reaktiven Verhaltens benötigt werden.

Diese Möglichkeiten zum Aufbau einer synthesefähigen RTL-Beschreibung werden im Rahmen der nachfolgenden Unterabschnitte anhand von Beispielen weiter ausgeführt.

3.1.1 Instanzen von Logikprimitiven und selbstdefinierten Modulen

Die Instanzierung von Entwurfselementen und deren Verdrahtung untereinander in Netzlistenform ist eine sehr strukturorientierte Beschreibungsform in Verilog. Sie kann bei Übereinstimmung der Entwurfselemente mit der Zieltechnologie direkt in eine Eingangsbeschreibung für die Layouterstellung umcodiert werden oder zur Vorbereitung einer Post-Layout-Simulation aus einem Layout extrahiert werden.

Einige generische Entwurfselemente der Logikebene, wie NAND-Gatter, Inverter oder Tri-State-Treiber, sind fest in der Sprache verankert und können ohne zusätzliche Definitionen instanziert werden. Diese Logikprimitive sind:

- Verknüpfungen mit 2 oder mehr Eingängen der logischen Funktionen `and`, `nand`, `or`, `nor`, `xor` und `xnor`
- Treiber (`buf`) und Inverter (`inv`), wobei diese auch Tri-State-Fähigkeit besitzen können (`bufif0`, `bufif1`, `notif0` oder `notif1`)

Beispiel 3.1 Die Boolesche Funktion $Y = (A \wedge B \wedge C) \vee D$ läßt sich als Netzliste von Logikprimitiven entsprechend Bild 3.1 in Verilog beschreiben. Der Netzliste ist als Vergleich eine Schaltplandarstellung der Gatter und ihrer Verbindungen gegenübergestellt.

```
module beispiel (Y, A, B, C, D);
  output Y;
  input  A, B, C, D;

  wire   Y, A, B, C, D, E;

  and U1 (E, A, B, C);
  or  U2 (Y, D, E);

endmodule
```

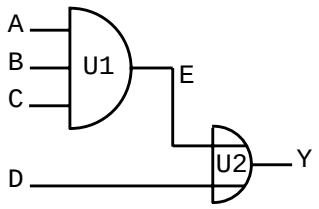


Bild 3.1: Instanzen von Verilog-Primitiven und korrespondierender Schaltplan

Neben Logikprimitiven können in Modulen auch andere, selbstdefinierte Module instanziert werden. Diese können wie in Beispiel 3.1 auf der Basis von Instanzen als Netzlisten aufgebaut werden oder mit Anweisungen in Continuous-Assignments bzw. `always`-Blöcken.

Die Logikprimitive sind permanent aktiv und reagieren auf jede Änderung ihrer Eingangswerte mit einer parametrisierten Verzögerung. Die Darstellung der Zeit ist für Logikprimitive auf Laufzeiten von Gattern beschränkt, welche die Auflösungsgrenze der für die RTL-Synthese spezifizierbaren Elemente bilden. Die Datenverarbeitung erfolgt auf der Ebene von Bits durch logische Verknüpfungen, wobei im Fall von Tri-State-Ausgängen die Ebene der elektrischen Signale berührt wird.

Bei Instanzen selbstdefinierter Module hängen die Darstellung der Zeit, die Hierarchieebene der Entwurfselemente und die Auflösung der Struktur von dem Inhalt dieser Module ab, so daß ohne dessen Kenntnis keine Aussage über diese Eigenschaften möglich ist.

3.1.2 Continuous-Assignments

Die kompakteste Beschreibung kombinatorischer Logik mit Anweisungen ist in Verilog mit Continuous-Assignments möglich. Diese reagieren ebenso wie die Logikprimitive auf jede Veränderung ihrer Eingangswerte und ersparen damit die explizite Aufstellung einer Sensitivitätsliste. Sie bieten gleichzeitig den Komfort der Formelschreibweise für Berechnungen. Das Ziel einer kontinuierlichen Zuweisung ist immer ein Netz, wobei die Zuweisung in der Deklaration des Netzes definiert wird oder über eine zusätzliche `assign`-Definition.

Beispiel 3.2 Die Boolesche Funktion $Y = (A \wedge B \wedge C) \vee D$ aus Beispiel 3.1 ist mit einem Continuous-Assignments entsprechend Bild 3.2 beschreibbar, wobei die Form der `assign`-Definition verwendet wird.

```
module beispiel (Y, A, B, C, D);
  output Y;
  input  A, B, C, D;

  wire   Y, A, B, C, D;

  assign Y = (A & B & C) | D;

endmodule
```

Bild 3.2: Continuous-Assignment mit `assign`-Definition

Die Zeitdarstellung erfolgt bei Continuous-Assignments in Form von Laufzeiten für die Zuweisungsoperation. Die Verhaltensdarstellung umfaßt neben Booleschen Funktionen arithmetische Berechnungen und Algorithmen (in Unterfunktionen), wobei die Struktur bis zu den logischen bzw. arithmetischen Operationen und ihren Operanden aufgegliedert wird. Als Daten können Bits, Bitvektoren oder Bitfelder mit begrenzten Signaleigenschaften (Tri-State) verarbeitet werden.

3.1.3 Anweisungsblöcke mit dem `always`-Konstrukt

Die Modellierung von Logik mit Anweisungen innerhalb eines `always`-Blockes bietet im Vergleich zur Instanzierung von Logikprimitiven oder Continuous-Assignments eine größere Auswahl von Logikelementen. Neben kombinatorischer Logik sind zusätzlich sequentielle Logikelemente beschreibbar, wie Flip-Flops oder Latches. Die Differenzierung zwischen kombinatorischer und sequentieller Logik erfolgt bei der Modellierung durch die Sensitivitätsliste des Blockes und die Zugriffsbedingungen auf Register. Die kombinierbaren Grundformen für diese Logiktypen werden zunächst einzeln vorgestellt.

Kombinatorische Logik: Eine notwendige Voraussetzung für die Darstellung kombinatorischer Berechnungen und ihrer Abhängigkeit von allen Eingangswerten in einem `always`-Block ist dessen vollständige Sensitivitätsliste. Die Liste ist vollständig, wenn in ihr alle im Block gelesenen Netze und Register frei von Flankenangaben wie `posedge` oder `negedge` vorkommen. Eine Ausnahme bilden Register, die in dem `always`-Block einen Wert zugewiesen bekommen, bevor sie gelesen werden. Diese brauchen nicht in der Sensitivitätsliste aufgeführt zu werden.

Die Zuweisungen an ein Register müssen außerdem in jedem alternativen Ausführungspfad eines `always`-Blockes stattfinden. Werden einem Register nur unter bestimmten Bedingungen Werte zugewiesen, so ist dieses nicht der Ausgang einer kombinatorischen Logik, sondern ein pegelgesteuertes Speicherelement.

Beispiel 3.3 Die Boolesche Funktion $Y = (A \wedge B \wedge C) \vee D$ aus Beispiel 3.1 ist in einem `always`-Block mit Zuweisungen wie in Bild 3.3 als kombinatorische Logik darstellbar. Die Sensitivitätsliste nach dem `@`-Operator ist vollständig, da sie alle in dem Block verarbeiteten Daten referenziert (Signale `A`, `B`, `C` und `D`), die nicht in dem Block selbst gesetzt werden. Das Register `TMP` wird vor seiner Verarbeitung mit einem Wert belegt und daher in der Sensitivitätsliste nicht benötigt. Die Verilog-Register `Y` und `TMP` stellen wegen ihrer kombinatorischen Abhängigkeit von den Eingangssignalen keine sequentielle Logik dar, auch wenn der Datentyp `reg` dies suggeriert. Sie sind nur Hilfsvariablen für die Beschreibung der Funktion mit den prozeduralen Anweisungen des `always`-Prozesses.

```
module beispiel (Y, A, B, C, D);
  output Y;
  input  A, B, C, D;

  reg    Y;
  wire   A, B, C, D;

  reg    TMP;

  always @(A or B or C or D) begin
    TMP = A & B & C;
    Y   = TMP | D;
  end

endmodule
```

Bild 3.3: Beschreibung kombinatorischer Logik in einem `always`-Block

Die Eigenschaften dieser Modellierungsform für kombinatorische Logik sind denen von Continuous-Assignments ähnlich. So umfaßt die Verhaltensdarstellung ebenfalls Boolesche Funktionen, arithmetische Berechnungen und Algorithmen. Die Strukturauflösung der Beschreibung reicht wiederum bis zu logischen bzw. arithmetischen Operationen und ihren Operanden. Es können Bits, Bitvektoren oder Bitfelder mit Tri-State-Fähigkeit verarbeitet werden. Die Zeitdarstellung in

Form der Verzögerung # für die Zuweisungsoperation = ist jedoch zu vermeiden, da sie den Anweisungsprozeß des `always`-Konstruktes blockiert und die Reaktion auf sich ändernde Eingangswerte verhindert.

Die Verwendung der nichtblockenden Zuweisung <= löst das Problem der Prozeßblockierung, ist wegen ihres verzögernden Zuweisungsverhaltens aber nur als abschließende Zuweisung einer Berechnung sinnvoll. Die Laufzeitangabe erfolgt in diesem Fall für den gesamten Berechnungsablauf. Alternativ kann die Angabe der Laufzeiten getrennt von der Logik in einem `specify`-Block erfolgen.

Pegelgesteuerte Speicherelemente: Werden Zuweisungen an ein Register in einem `always`-Block mit vollständiger Sensitivitätsliste nicht in jedem möglichen Ausführungspfad vorgenommen, so wird mit diesem Register ein pegelgesteuertes Speicherelement (*Latch*) modelliert. Ausnahmen stellen als Hilfsvariablen benutzte Register dar, die außerhalb des Blockes nicht gelesen werden und als Ausgänge kombinatorischer Logik verwendet werden. In dem Block sind zudem kombinatorische Berechnungen entsprechend Beispiel 3.3 durchführbar.

Beispiel 3.4 Der `always`-Block in Bild 3.4 modelliert ein Latch für den Ausgang `Q`, welches Daten vom Eingang `D` übernimmt, wenn das Freigabesignal `E` den Wert 1 führt. Hat `E` den Wert 0, so erfolgt keine Zuweisung an `Q`.

```
always @ (D or E) begin
  if (E == 1)
    Q <= D;
end
```

Bild 3.4: Latch-Modellierung in einem `always`-Block

Zusätzlich zur Modellierung kombinatorischer Logik und ihren verschiedenen Verhaltensdarstellungen wird durch dieses Konstrukt die Speicherung von Daten eingeführt. Die Strukturauflösung umfaßt neben logischen und arithmetischen Operatoren und ihren Operanden einzelne Speicherelemente, deren Reaktionszeit in Form von Zuweisungsverzögerungen dargestellt wird. Daten werden als Bits, Bitvektoren oder Bitfelder mit Tri-State-Fähigkeit beschrieben.

Als Zuweisungsoperation für Latch-Speicherelemente ist <= zu bevorzugen, um abweichende Simulationsergebnisse nach der Synthese auszuschließen. Für Zuweisungen an Hilfsvariablen oder kombinatorische Ergebnisse gelten die selben Bedingungen hinsichtlich Zuweisungsoperationen und ihren Zeitangaben, wie für vollständig kombinatorische `always`-Blöcke.

Flankengesteuerte Speicherelemente: Durch die gezielte Reaktion auf bestimmte Signalflanken in der Sensitivitätsliste sind flankengesteuerte Speicherelemente (*Flip-Flops*) mit `always`-Blöcken modellierbar. Hierzu wird das Taktsignal der zu modellierenden Flip-Flops in der Sensitivitätsliste durch eine der Flankenangaben `posedge` (für Reaktion auf steigende Taktflanken) oder `negedge` (für Reaktion auf fallende Taktflanken) eingeleitet. Außer dem Takt dürfen in der Sensitivitätsliste nur Signale aufgeführt werden, die eine zum Takt asynchrone Initialisierung der

Flip-Flops mit konstanten Werten auslösen. Diese für das asynchrone Löschen (*Reset*) und Setzen (*Set*) von Flip-Flops benutzten Signale müssen ebenfalls durch Flankenangaben eingeleitet werden, wobei die Flankenrichtung den Beginn der Initialisierungsphase kennzeichnet. Die Verarbeitung dieser Signale erfolgt durch `if`-Abfragen auf ihre aktiven Werte, welche in der ersten Zeile des `always`-Blockes beginnen und entsprechend der Signalpriorität verschachtelt sind. Der Takt wird demjenigen Signal der Sensitivitätsliste zugeordnet, das in dem Block nicht gelesen wird.

Beispiel 3.5 In Bild 3.5 wird ein Flip-Flop in einem `always`-Block modelliert. Bei der fallenden Flanke des Taktes `C` werden die Daten vom Eingang `D` in das Register `Q` übernommen, wenn keine der asynchronen, 1-aktiven Initialisierungen `S` (Set) und `R` (Reset) gesetzt ist und die synchrone Freigabe des Flip-Flops `E` (Enable) ihren aktiven Wert 1 führt. Das Reset-Signal hat aufgrund seiner Position in der `if-else`-Hierarchie Priorität über das Set-Signal.

```
always @ (negedge C or posedge S or posedge R) begin
  if (R == 1'b1)        // Reset
    Q <= 1'b0;
  else
    if (S == 1'b1)      // Set
      Q <= 1'b1;
    else
      if (E == 1'b1)    // Enable
        Q <= D;
end
```

Bild 3.5: Modellierung eines Flip-Flops

In einem taktflankenaktivierten `always`-Block modelliert jedes Register, das Ziel der Zuweisungsoperation `<=` ist, ein Flip-Flop, da hierbei der neu zugewiesene Registerwert unabhängig von seinen Quelloperanden bis zur nächsten Taktflanke gehalten werden muß. Die Zuweisungsoperation `=` modelliert nur dann ein Speicherelement, wenn die Zielvariable außerhalb des Blockes gelesen wird oder bis zum nächsten Blockdurchlauf gehalten werden muß, also keine Hilfsvariable einer kombinatorischen Berechnung ist. Ähnlich den Latches ist als Zuweisungsoperation für Flip-Flops `<=` zu bevorzugen, um Simulationsabweichungen nach der Synthese auszuschließen. Für Zuweisungen an Hilfsvariablen sollten aus demselben Grund keine Laufzeiten angegeben werden.

Durch die flankengesteuerte Blockaktivierung wird im Vergleich mit Latches zusätzlich zu den Laufzeiten die getaktete Zeitdarstellung eingeführt, so daß neben den Zuweisungsverzögerungen auch die Sensitivitätsliste zeitliche Aspekte beschreibt. Neben kombinatorischer Logik und ihren Verhaltensdarstellungen ist die Speicherung von Daten modellierbar. Die Strukturauflösung umfaßt logische und arithmetische Operatoren und ihre Operanden sowie Speicherelemente. Daten werden wiederum als Bits, Bitvektoren oder Bitfelder mit Tri-State-Fähigkeit verarbeitet.

3.2 Zusammensetzung der Register-Transfer-Konstrukte

Die im vorigen Abschnitt vorgestellten Verilog-Konstrukte können nach der RTL-Verilog-Syntax erweitert und zu größeren Beschreibungen zusammengesetzt werden. Dabei sind mehrere Einschränkungen zu beachten, die synthesegeeignete Modelle sicherstellen und fehlerhafte Unterschiede zwischen den Ergebnissen von Simulationen vor und nach der Synthese verhindern. Die Einschränkungen betreffen:

- Anweisungen in `always`-Blöcken, Tasks und Funktionen
- Hochohmige Werte (Tri-States)
- Don't-Care-Werte

Auf diese Bereiche wird im Rahmen der nachfolgenden Unterabschnitte unter Verwendung von Beispielen weiter eingegangen.

3.2.1 Anweisungen in `always`-Blöcken, Tasks und Funktionen

Im Rahmen der prozeduralen Anweisungsblöcke von `always`-Prozessen, Tasks und Funktionen sind bis auf wenige Ausnahmen die selben Anweisungen verwendbar. Neben den bereits vorgestellten Zuweisungsarten = und <= sind mehrere Selektionen (`if-else`, `case`, `casex`, `casez` und `?-:`) und `for`-Schleifen mit konstanten Laufgrenzen in synthesegeeignetem Verilog-Code erlaubt. Der Aufruf von Tasks ist auf `always`-Blöcke und Tasks beschränkt, Funktionen dürfen keine Tasks benutzen. Die Verzögerung # ist nur in `always`-Blöcken möglich, ebenso wie die Synchronisation @, die als weitere Einschränkung nur direkt nach `always` stehen darf.

Zuweisungen: Die Anwendung der Zuweisungen = und <= wurden bereits im vorangegangenen Abschnitt im Zusammenhang mit den Konstrukten für Register-Transfer-Elemente vorgestellt. Tasks und Funktionen sind aufgrund ihrer vom Aufrufer abhängigen Aktivierung nur für kombinatorische Zwischenrechnungen zu verwenden, um Inkonsistenzen in der Modellierung von RT-Elementen sicher auszuschließen. Sie sollten daher durchgehend mit der Zuweisung = modelliert werden.

Eine von den vorgestellten Konstrukten abweichende Verwendung der zwei Zuweisungsarten kann zu fehlerhaften Unterschieden zwischen den Simulationsergebnissen eines Entwurfs vor und nach der Synthese führen. Die Verwendung von <= in kombinatorischen Zwischenrechnungen kann aufgrund der Zuweisungsverzögerung an das Ende eines Simulationszeitschrittes Datenabhängigkeiten zeitlich so verschieben, daß mehrere Blockaktivierungen für die korrekte Berechnung erforderlich sind. Ist der `always`-Block anders als die dauerhaft aktiven Logikprimitive nicht für das verzögerte Zuweisungsergebnis sensitiv, so führt die Simulation vor der Synthese unvollständige Berechnungen aus.

Die Zuweisung = kann für Speicherelemente zu fehlerhaften Unterschieden zwischen den Simulationsergebnissen vor und nach der Synthese führen. Werden in getrennten `always`-Blöcken mit = modellierte Speicherelemente zu einer Kette wie in Bild 3.6 verschaltet, so hängt bei verzögerungsfreiem Transfer der Daten

von D nach Q das Ergebnis der Simulation vor der Synthese vom Simulator und der von ihm gewählten Reihenfolge der Blockbearbeitung ab. Der Grund hierfür ist eine zeitliche Konkurrenz der Daten- und der Taktänderung, eine *Race-Condition*.

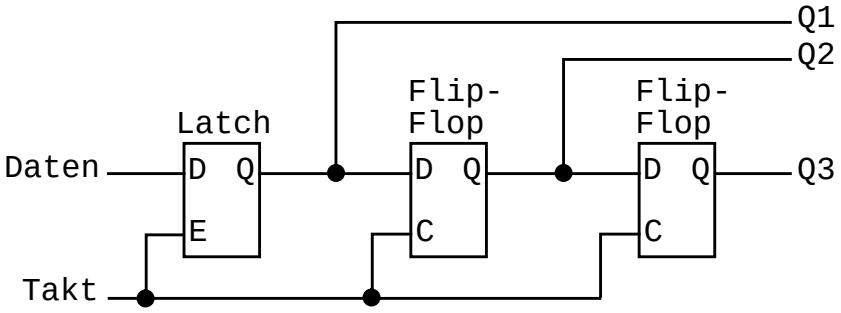


Bild 3.6: Speicherelemente, bei deren Modellierung mit = ein Race entsteht

Als Folge sind Zuweisungen auf ein Register innerhalb eines always-Blockes nur mit einer der beiden Zuweisungsarten ausführbar. Zuweisungen aus mehreren always-Blöcken auf ein Register führen unabhängig von der Zuweisungsart wiederum zu einem Race. Bei der Erweiterung und Kombination der genannten Verilog-Konstrukte sind derartige Modellierungen als Fehler anzusehen und nicht zu verwenden.

Selektionen: Neben der in Beispiel 3.5 benutzten if-else-Selektion sind in Verilog die case-Selektion und ihre Varianten casex und casez, sowie die Selektion ?-: möglich. Dabei ist (bedingung) ? anweisung1 : anweisung2 gleichwertig mit if (bedingung) anweisung1 else anweisung2, die ?-:-Selektion also nur eine Kurzschreibweise für if-else. Die case-Selektion und ihre Varianten bieten eine kompakte, tabellenorientierte Schreibweise für die Abfrage von Daten mit mehreren Bit Breite und die Auswahl der auszuführenden Anweisungen. Sie sind durch verschachtelte if-else-Selektionen gleichwertig zu ersetzen.

Abgesehen von der Verwendung der if-else-Abfrage bei flankengesteuerten Speicherelementen für asynchrone Steuersignale sind die mit den verschiedenen Selektionen beschreibbaren Logiken gleich. Je nach Anordnung, Verschachtelung und Überschneidung von Auswahlfällen und davon abhängigen Zuweisungen entsprechen die Selektionen Multiplexern, Prioritäts-Encodern oder Latches mit Freigabe- und Datenauswahllogik.

Bei den case-Varianten casex und casez kann es im Gegensatz zu den übrigen Selektionsanweisungen zu unterschiedlichen Simulationsergebnissen vor und nach der Synthese kommen. Bitpositionen mit den Werten Z bei casez bzw. X oder Z bei casex werden bei der Auswahl nicht beachtet (*Don't-Care*), unabhängig davon, ob diese im abzufragenden Bitvektor oder im Auswahlvektor vorkommen.

Dies kann in der Simulation eines Entwurfs vor seiner Synthese dazu führen, daß trotz eines abgefragten Bitvektors mit undefinierten oder hochohmigen Bits eine Selektion definiert arbeitet. Auf diese Weise kann auch eine unvollständig initialisierte Schaltung in der Simulation korrekt einschwingen. In der Simulation

nach der Synthese werden jedoch nur noch die im Auswahlvektor als Don't-Care markierten Bits nicht beachtet. Undefinierte bzw. hochohmige Bits im abgefragten Bitvektor, die nicht gleichzeitig Don't-Cares des Auswahlvektors sind, führen dann im Gegensatz zur Simulation vor der Synthese zu undefinierten Selektionen.

Beispiel 3.6 Die `casex`-Selektion in Bild 3.7 liefert in einer Simulation vor ihrer Synthese auch definierte Werte am Ausgang `Y`, wenn der Eingang `B` den Wert `X` oder `Z` führt. Verilog durchsucht in diesem Fall die `case`-Tabelle sequentiell nach dem ersten zu `A` passenden Eintrag und beachtet `B` nicht. In einer Simulation nach der Synthese ist bei undefiniertem `B` und dem Wert 0 auf `A` der Ausgang `Y` nicht definiert, da er gleichberechtigt 0 oder 1 sein könnte.

```
always @(A or B) begin
  casex({A, B})
    2'b00: Y = 1'b1;
    2'b01: Y = 1'b0;
    2'b10: Y = 1'b1;
    2'b11: Y = 1'b1;
  endcase
end
```

Bild 3.7: Selektion mit `casex` und möglicher Simulationsinkonsistenz

Trotz der Möglichkeiten, abweichende Simulationsergebnisse hervorzurufen oder undefinierte bzw. hochohmige Werte zu verbergen, bieten `casex` und `casez` Vorteile, die ihre Verwendung gegenüber den übrigen Selektionen rechtfertigen. Sie erlauben in der tabellarischen Auflistung neben `X` bzw. `Z` die Benutzung des Platzhalters `?` für Don't-Cares im Auswahlvektor und ermöglichen in einigen Fällen eine noch kompaktere Darstellung, als die normale `case`-Anweisung.

Die Benutzung von `casez` birgt im Vergleich zu `casex` geringere Risiken einer Simulationsinkonsistenz, da hier nur der Wert `Z` als Don't-Care interpretiert wird. Aufgrund der Vorinitialisierung von Registern und Verbindungen in Simulationen auf den Wert `X` ist die Wahrscheinlichkeit von Simulationsinkonsistenzen bei `casex` besonders für unvollständige Initialisierungen höher. Die Benutzung von `casez` ist somit der von `casex` vorzuziehen [MilCum99].

Reguläre Vervielfältigung mit `for`-Schleifen: Die Verwendung von `for`-Schleifen in RTL-Verilog dient zur regulären Vervielfältigung kombinatorischer Logik. Während `for`-Schleifen in Simulationen sequentiell ausgeführt werden, ergibt ihre Synthese parallel aktive Baugruppen für jeden Schleifendurchlauf. Die Schleifengrenzen sind auf konstante Werte beschränkt und die Laufvariablen nur für die Simulation als Register vorhanden, da bei der Synthese eine eigene Baugruppe für jeden erreichbaren Wert der Laufvariable angelegt wird. Die Schleifenkontrolle ist auf fest vorgegebene Grundformen beschränkt.

Beispiel 3.7 Die `for`-Schleife in Bild 3.8 modelliert einen Ripple-Carry-Addierer für 3 Bit Breite Daten `A` und `B` mit einem 1 Bit Carry-Eingang `CI`. Als Ergebnis wird eine 3 Bit Summe auf `S` und ein 1 Bit Übertrag auf `CO` ausgegeben. Innerhalb

der Schleife wird ein 1 Bit Volladdierer modelliert, der durch die Schleife 3-fach vervielfältigt wird, wobei eine Übertragskette entsteht.

```
wire [2:0] A, B;
wire       CI;
reg  [2:0] S;
reg        CO;
integer    I;

always @(A or B or CI) begin
  CO = CI;
  for (I = 0; I <= 2; I = I + 1) begin
    S[I] = A[I] ^ B[I] ^ CO[I];
    CO   = (A[I] & B[I]) |
           (A[I] & CO) |
           (B[I] & CO);
  end
end
```

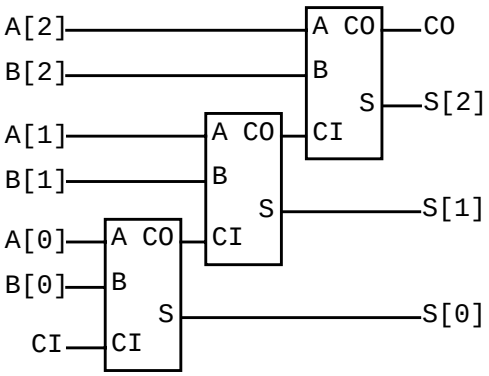


Bild 3.8: Modellierung eines 3 Bit Addierers mit regulärer Vervielfältigung

Tasks und Funktionen: Mit Tasks und Funktionen können Anweisungsfolgen für kombinatorische Berechnungen in RTL-Verilog aus Continuous-Assignments oder `always`-Blöcken ausgelagert werden. Innerhalb von Tasks und Funktionen gelten mit einer Ausnahme dieselben Modellierungsrichtlinien, wie für kombinatorische Logik in `always`-Blöcken, die Angabe von Laufzeiten ist hier nicht zulässig. Zwischen Tasks, Funktionen, Continuous-Assignments und `always`-Blöcken besteht außerdem eine feste Aufrufhierarchie, wie Bild 3.9 zeigt.

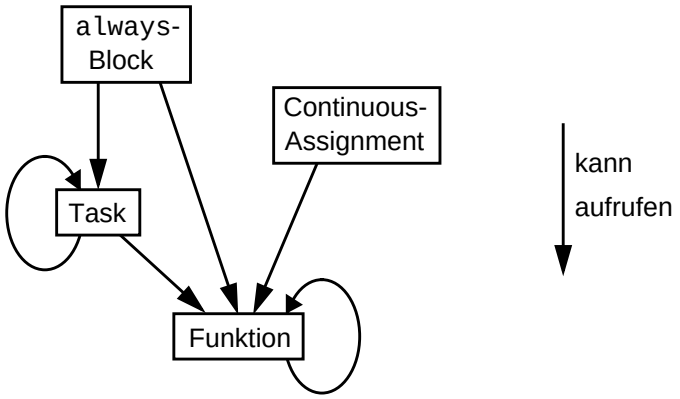


Bild 3.9: Aufrufhierarchie zwischen kombinatorischen Berechnungen

Der Aufruf von Tasks durch Tasks bzw. Funktionen durch Funktionen ist nur dann erlaubt, wenn es sich nicht um einen rekursiven Aufruf einer Task bzw. Funktion durch sich selbst handelt.

Die Verzögerungsanweisung #: Die Angabe von Laufzeiten mit der Verzögerung # ist im Vergleich zu `specify`-Blöcken eine einfache Möglichkeit zur direkten Zeitspezifikation innerhalb der bereits vorgestellten RTL-Verilog-Konstrukte. Sie

ist speziell bei Testrahmen für die Erzeugung von Signalverläufen der Stimuli unentbehrlich. Ihre Verwendung erfolgt als einzelne Anweisung, als Zusatz zu einer Zuweisung oder als Parameter bei der Instanzierung von Logikprimitiven bzw. der Deklaration von Netzen.

Aufgrund der Nichtbeachtung von Laufzeitangaben bei der RTL-Synthese bergen Laufzeitangaben mehrere Risiken für fehlerhafte Unterschiede zwischen Simulationen vor und nach der Synthese. So können sie als einzelne Anweisungen oder Zusatz der Zuweisung = den sie umgebenden `always`-Block in Simulationen blockieren, so daß dieser nicht auf Änderungen von Signalen der Sensitivitätsliste reagieren kann. Dies führt im Vergleich zu der Simulation nach der Synthese zu einem fehlerhaften reaktiven Verhalten vor der Synthese.

In jeder ihrer Anwendungsformen können Verzögerungen zur Modellierung asynchroner Protokolle von Signalen verwendet werden. Da die Abstimmung der Signale mit # bei der RTL-Synthese verlorengeht, können diese Protokolle nicht korrekt umgesetzt werden. Dies kann zu Race-Conditions nach der Synthese und somit zu undefinierten Modellzuständen führen.

Ein weiterer möglicher Fehlermechanismus ist bei kombinatorischen `always`-Blöcken die Verzögerung von Hilfsberechnungen, die mehrere Blockdurchläufe für vollständige Berechnungen erfordern kann. Bei fehlender Sensitivität des Blockes für die verzögerten Zwischenergebnisse bleiben die Berechnungen vor der Synthese unvollständig.

Wegen des Risikos, mit Verzögerungen nicht korrekt synthetisierbare Modelle in RTL-Verilog zu schreiben, sollten sie in allen genannten Grundkonstrukten für Register-Transfer-Elemente vermieden werden.

Die Synchronisationsanweisung `@`: Das reaktive Verhalten von Anweisungen wird in Verilog mit der Synchronisationsanweisung `@` beschrieben, die in RTL-Verilog auf die bereits vorgestellten Grundformen und die Plazierung nach dem Schlüsselwort `always` beschränkt ist.

Bei der Modellierung kombinatorischer Logik oder Latches kann auch diese Anweisung zu Simulationsfehlern vor der Synthese führen, wenn die Sensitivitätsliste nicht alle für die Aktualisierung der modellierten Logik relevanten Signale enthält. Neben dem für die Verzögerung # bereits beschriebenen Fehlerszenario kann auch die nichtblockende Zuweisung <= durch ihre Verzögerung an das Ende eines Simulationszeitschrittes ein ähnliches Fehlerbild bewirken.

Aus diesem Grund sollte die nichtblockende Zuweisung <= möglichst nur für Speicherelemente benutzt werden. Die Sensitivitätsliste sollte neben allen von außerhalb des Blockes beschriebenen Signalen und Variablen auch diejenigen beinhalten, die innerhalb des Blockes verzögert zugewiesen werden und gelesen werden.

3.2.2 Hochohmige Werte (Tri-States)

Die Benutzung von hochohmigen Werten wird in RTL-Verilog für die Modellierung von Tri-State-Treibern benutzt und ist eine Alternative zur Instanzierung von

Logikprimitiven mit Tri-State-Ausgang. Je nach umgebender Modellierung erzeugt die Synthese die zur Ansteuerung der Treiber nötige kombinatorische oder sequentielle Logik.

Ein wesentliches Element für die Modellierung der Treibersteuerung ist die Verwendung einer Selektion in einem `always`-Block oder Continuous-Assignment, welche die Bedingungen für den hochohmigen Zustand und das Treiben von Daten festlegt.

Abhängig von dem benutzten Modellierungskonstrukt und der beabsichtigten Datenrichtung gibt es mehrere Beschreibungsformen für Tri-State-Treiber, die entsprechend der übrigen Logikmodellierung erweiterbar sind (Beispiel 3.8).

Werden hochohmige Daten in `casex`- oder `casez`-Selektionen abgefragt, so sind wegen ihrer Don‘t-Care Behandlung unterschiedliche Simulationsergebnisse vor und nach der Synthese möglich. Auch Wertänderungen zwischen hochohmig und 0 bzw. 1 erfolgen nach der Synthese aufgrund von kapazitiver Speicherung oder fehlender Signalflankensteilheit anders, als von Simulationen vor der Synthese her gewohnt. Bei der Verarbeitung möglicherweise hochohmiger Signale und insbesondere der Sensitivität auf Flanken solcher Signale ist somit Vorsicht geboten.

Beispiel 3.8 Die Verilog-Fragmente in Bild 3.10 zeigen mehrere Möglichkeiten zur Modellierung von Tri-State-Treibern. Die ausschließliche Benutzung von Registerzugriffen aus einem `always`-Block (`TriState1`) erlaubt nur Ausgabetreiber. Durch kombinierte (`TriState2`) oder ausschließliche Benutzung von Continuous-Assignments sind bidirektionale Verbindungen und modulinterne Bussysteme modellierbar.

```
// TriState1
input in, enable;
output driver;

reg driver;

always @(in or enable)
  if (enable == 1‘b1)
    driver = in;
  else
    driver = 1‘bZ;
```

```
// TriState2
inout data;
input write, enable;
reg   driver, read;
wire  data = driver;

always @(write or enable or data)begin
  read = data;
  if (enable == 1‘b1) driver = write;
  else driver = 1‘bZ;
end
```

Bild 3.10: Varianten zur Modellierung von Tri-State-Treibern

3.2.3 Don‘t-Care-Werte

Zuweisungen des Wertes `X` an ein Register oder Netz stehen in einer Simulation für ein undefiniertes Datum und werden dort als Reaktion auf nicht vorgesehene Eingangsbelegungen von Schaltungsteilen benutzt. Für die RTL-Synthese ist ein `X` auf der rechten Seite einer Zuweisung ein Don‘t-Care-Wert, d.h. der Wert ist ohne Bedeutung und das Zuweisungsergebnis beliebig wählbar.

Für die Synthese bietet die Benutzung von Don't-Care-Werten Möglichkeiten zur Optimierung von Logikstrukturen, indem beispielsweise nicht vorgesehene Eingangsbelegungen einer der definierten Ansteuerungen zugeordnet werden und die Anzahl der Gesamtbelegungen verringert wird.

Die Verwendung von Don't-Care-Werten führt zu abweichendem Logikverhalten vor und nach der Synthese. Werden Don't-Care-Zuweisungen durch entsprechende Stimuli aktiviert, so führt dies vor der Synthese zu X als Ergebnis, während nach der Synthese definierte Binärwerte berechnet werden.

Das Problem und der Schlüssel zu seiner Vermeidung liegt in diesem Fall bei den Stimuli, welche die Logik gezielt außerhalb ihrer funktionalen Spezifikation ansprechen. Nur für Stimuli, die zu definierten Simulationsergebnissen vor einer Synthese führen, können dieselben Ergebnisse auch nach der Synthese erwartet werden.

3.3 Optimierungsmöglichkeiten der RTL-Synthese

Bei der RTL-Synthese gibt es während der in den Grundlagen beschriebenen Umsetzung des Entwurfs in eine Gatternetzliste der Zieltechnologie verschiedene Möglichkeiten zur Optimierung des Ergebnisses. Diese Optimierungen können durch Optionen bei der Steuerung des Syntheseprogramms gewählt werden und sind in gewissem Umfang durch die Modellierung unterstützbar. Sie unterteilen sich in erster Linie nach den Optimierungszielen für den Entwurf, die parallel mit abgestufter Priorität untereinander verfolgt werden können. Erst in zweiter Linie sind die angewandten Techniken und Algorithmen von Bedeutung. Als Ziele für die Optimierungen der Synthese mit dem Synopsys Design-Compiler sind möglich:

- Schaltungsgeschwindigkeit (Taktperiode oder Laufzeiten)
- Schaltungsgröße (Fläche, Gate-Count oder FPGA-Logikblöcke)
- elektrische Leistungsaufnahme
- Testbarkeit mit automatisch generierter Scan-Testlogik
- Treiber- und Lastenoptimierung
- Taktnetzoptimierungen

Allerdings sind nur die beiden Optimierungsziele Schaltungsgeschwindigkeit und Schaltungsgröße für alle Schaltungstechnologien anwendbar, weswegen es die einzigen in dieser Arbeit weiter betrachteten Ziele sind. Diese Optimierungsziele und die Möglichkeiten sie zu erreichen sind zudem nicht auf den Design-Compiler beschränkt. Die Optimierungen werden vorgenommen durch:

- Auswahl zwischen alternativen Komponenten beim Mapping
- Aufbau verschiedener Logikstrukturen (z.B. DNF oder kaskadierte Logik)
- Mehrfachverwendung von arithmetischen Operatoren
- Entfernung von Redundanzen

Außerhalb der Optimierungsmöglichkeiten der RTL-Synthese liegen Änderungen an der Register-Transfer-Struktur aus kombinatorischer und sequentieller Logik und Eingriffe in das vorgegebene Taktungsschema.

Die Syntheseoptionen des Design-Compilers und Vorgabemöglichkeiten für Geschwindigkeit und Größe werden hier kurz vorgestellt. Nähere Informationen zu den Optimierungsmöglichkeiten und -optionen sind in der Dokumentation des Design-Compilers [SynDCR97] zu finden.

Geschwindigkeitsvorgaben: Die Optimierung der Schaltungsgeschwindigkeit ist im Rahmen einzelner Module durch Vorgabe der Periodendauer für einen Takt und maximaler Laufzeiten für Signalpfade steuerbar. Übermäßige Optimierungen von kombinatorischer Logik zwischen Registern, für deren Einschwingen mehrere Takte erlaubt sind, können durch Markierung der betroffenen Signalpfade als *Multicycle Paths* verhindert werden.

Während der Schaltungssynthese werden durch die Optimierungsverfahren die kombinatorischen Schaltungsanteile so implementiert, daß diese Vorgaben im Rahmen der vorhandenen Zusatzbedingungen bestmöglich erfüllt werden.

Größenvorgaben: Je nach verwendeter Zieltechnologie ist für jedes Modul eine maximale Größe der Implementierung als Fläche, Gate-Count, oder Anzahl der Zellen bzw. Logikblöcke spezifizierbar. Dieser Vorgabewert wird bei der Synthese angestrebt, aber der Geschwindigkeitsoptimierung untergeordnet.

Auflösung von Modulgrenzen (ungroup): Diese Optimierungsoption ermöglicht bei der Synthese eines Moduls die Auflösung seiner Grenzen zu Untermodulen. Durch den Wegfall der Schnittstellen und der Logikgruppierungen in Modulen, die sonst von der Synthese unangetastet bleiben, können kombinatorische Logiken vereint und von Schnittstellenredundanzen befreit werden.

Der Zuwachs an zu betrachtender Logik für einen Syntheselauf führt zu einem exponentiell größeren Bedarf an Speicher und Rechenzeit, so daß der Auflösung von Modulgrenzen während der Synthese technische Grenzen gesetzt sind.

Bildung von Disjunktiven Normalformen (flatten logic): Bei dieser Optimierung wird die kombinatorische Logik eines Moduls durch Disjunktive Normalformen (DNF) mit ODER-Verknüpfungen von UND-Termen umgesetzt. Diese zweistufige Logikdarstellung ist für die Auffindung von redundanter Logik besonders geeignet, die mit schwerpunktmäßiger Betrachtung eines Ausgangs oder mehrerer Ausgänge entfernt werden kann.

Ein Nachteil dieser Optimierung ist jedoch die Auflösung der in RTL-Verilog beschriebenen Datenflußstrukturen. Insbesondere arithmetische Verknüpfungen werden durch die DNF-Umsetzung ineffizient, da ihre regulären Strukturen verloren gehen und sie damit nicht mehr optimal auf die Zielbibliothek abgebildet werden können.

Bildung kaskadierter Logik (structure logic): Durch mehrstufige Logik und die Mehrfachbenutzung gemeinsamer Logikterme für verschiedene Signalpfade wird bei dieser Optimierung die Menge an kombinatorischer Logik reduziert. Dabei

bleiben arithmetische Operatoren und durch die Modellierung vorgegebene Datenflüsse erhalten. Als Optionen dieser Optimierung können möglichst geringe Pfadlängen und die Anwendung Boolescher Optimierungen gewählt werden.

Diese Optimierung arbeitet der Bildung Disjunktiver Normalformen entgegen und sollte daher nicht gleichzeitig mit dieser aktiviert werden, auch wenn dies von der Auswahl her möglich ist.

Periphere Optimierungen (boundary optimization): Hierbei werden die Grenzen zu Untermodulen auf redundante Invertierungen und Verknüpfungen überprüft, die durch Invertierung der Schnittstellenpolarität oder das Verschieben einzelner Gatter über diese Grenzen beseitigt werden.

Erhaltung der Modulstruktur (don‘t touch): Diese Option nimmt ein Modul von weiteren Optimierungen jeder Art aus. Dies ist angebracht, wenn ein Modul durch gezielte Instanzierung von Logikprimitiven oder vorangegangene Optimierungen bereits den Zielvorgaben entspricht und ihnen durch weitere Arbeitsschritte nicht weiter angenähert werden kann.

Mehrfachnutzung arithmetischer Operationen (resource sharing): Benutzt ein `always`-Block in sich ausschließenden Anweisungspfaden einer Selektion dieselbe arithmetische Operation, so wird für diese dieselbe Hardware benutzt. Hierzu werden automatisch Multiplexer vor die Eingänge der Hardwareoperation gesetzt, um die jeweils benötigten Daten heranzuführen. Kostet dies zuviel Zeit, so kann das als Voreinstellung aktive Resource-Sharing des Design-Compilers über eine Option abgeschaltet werden.

3.4 Optimierungsmöglichkeiten bei der Modellierung in RTL-Verilog

Die Modellierung bietet im Vergleich zur RTL-Synthese mehr Möglichkeiten zur Optimierung eines Entwurfs für bestimmte Entwurfsziele. Während die RTL-Synthese auf die Optimierung kombinatorischer Schaltungsteile zwischen den von Speicherelementen gesetzten Grenzen beschränkt ist, bietet die Modellierung einen großen Freiraum für Optimierungen auf hohen Ebenen, bevor Grenzen durch Speicher und ihre Taktung festgelegt werden.

Neben alternativen Algorithmen bzw. Arbeitsschemata hat die Zuordnung von Berechnungen zu Takten, ihre sequentielle oder parallele Implementierung und die Aufteilung eines kombinatorischen Datenflusses in getaktete Einzelschritte (Pipelining) hohen Einfluß auf die erreichbaren Ergebnisse. Diese Optimierungen liegen bei der RTL-Synthese vollständig in der Hand des Entwicklers. Erst bei der im nächsten Kapitel vorgestellten High-Level-Synthese werden einige dieser Möglichkeiten durch das Synthesesystem genutzt.

4 High-Level-Synthese

Die Synthese von algorithmischen Verilog-Beschreibungen mit dem Behavior-Compiler und dessen High-Level-Syntheseverfahren sind das Thema dieses Kapitels. Hierbei werden zunächst der Aufbau und die Eigenschaften einer entsprechenden Entwurfsbeschreibung vorgestellt. Im Anschluß werden die bei der High-Level-Synthese ausgeführten Arbeitsschritte, die Möglichkeiten zu ihrer Beeinflussung und die vom Behavior-Compiler benutzte Hardwarearchitektur betrachtet.

Viele der hierbei angesprochenen High-Level-Syntheseverfahren sind typische Elemente von Synthesewerkzeugen für algorithmische Beschreibungen und nicht auf den Behavior-Compiler beschränkt. In Kombination mit anderen Verfahren und Architekturen für Spezialanwendungen entsteht die Vielzahl an Werkzeugen im High-Level-Synthesebereich, die sich durch die gemeinsamen Grundelemente von dem Bereich der Register-Transfer-Synthese abheben.

4.1 Verilog-Beschreibungen für den Behavior-Compiler

Die für die High-Level-Synthese mit dem Behavior-Compiler verwendeten Verilog-Modelle unterscheiden sich von den bereits vorgestellten RTL-Verilog-Modellen besonders deutlich durch ihre andere Beschreibungsform für `always`-Blöcke im obersten Modul der Entwurfshierarchie. Unterhalb dieses Moduls wird die Entwurfshierarchie durch RTL-Verilog-Module und Logikprimitive gebildet.

Die `always`-Blöcke im obersten Modul der Entwurfshierarchie unterscheiden sich durch ihre Prozeßsynchronisation von den Schemata für kombinatorische oder getaktete Logik. Die Synchronisation eines `always`-Blockes erfolgt ausschließlich auf einen Flankentyp eines Taktsignals, wobei eine oder mehrere `@`-Anweisungen zur Synchronisation innerhalb der Anweisungsfolge verwendet werden. In dem obersten Modul sind mehrere `always`-Blöcke und mehrere Takte benutzbar, aber nur jeweils ein Takt in einem Block.

Innerhalb eines Blockes sind neben den RTL-Modellierungsanweisungen `for`-Schleifen mit variablen Laufgrenzen und `while`-Schleifen zulässig. Zuweisungen an Variablen können blockend mit `=` oder nichtblockend mit `<=` erfolgen, an Port-Variablen jedoch nur nichtblockend. Die Verzögerung mit # ist nur im Rahmen der nichtblockenden Zuweisung erlaubt, als eigenständige Anweisung oder mit `=` aber im Gegensatz zu RTL-Beschreibungen nicht benutzbar.

Die Ein- und Ausgabe der algorithmischen Beschreibung erfolgt ausschließlich über unidirektionale Ports, bidirektionale Ports sind trotz der Verwendbarkeit von Tri-States nicht modellierbar. Der Grund hierfür liegt in der unterschiedlichen Zuordnung der Ein- und Ausgabezuweisungen zu Takten innerhalb der durch die angegebenen Taktsynchronisationen gesetzten Grenzen von Zeitschritten für die Anweisungen. Da abhängig von dem bei der Optimierung gewählten Zeitmodell für Ein- und Ausgaben und den Zielvorgaben für die anzustrebende Ausführungszeit ein Zeitschritt aus einem oder mehreren Einzeltakten bestehen kann, wird die für bidirektionale Kommunikation erforderliche zeitliche Abstimmung von Ein- und

Ausgaben beim Behavior-Compiler im allgemeinen nicht erreicht. Bidirektionale Ein- und Ausgaben sind in diesem Fall nur durch nachträgliche Einordnung des algorithmischen Entwurfs unter ein RTL-Modul möglich, das die Kommunikation auf der Basis unidirektionaler Daten- und Kontrollsignale bidirektional durchführt.

Einen Überblick über die durch den Behavior-Compiler verarbeitete Entwurfshierarchie und die in der algorithmischen Beschreibungsform verwendbaren Modellierungskonstrukte gibt Bild 4.1.

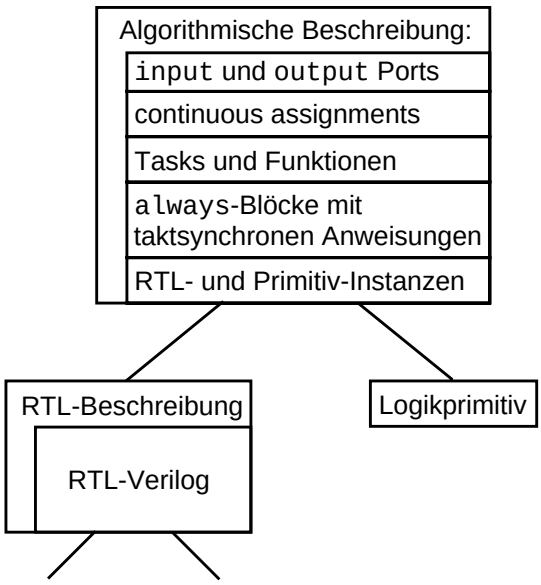


Bild 4.1: Entwurfshierarchie für den Behavior-Compiler

In einer High-Level-Beschreibung sind die Eigenschaften von Entwürfen weniger detailliert darstellbar als in einer RTL-Beschreibung. Die Darstellung des Entwurfsverhaltens erfolgt zwar auch mittels Logiktermen und Algorithmen, welche Bits, Bitvektoren oder Bitfelder mit Tri-State-Fähigkeit verarbeiten, Unterschiede bestehen jedoch bei der Darstellung der Zeit und der Auflösung der Struktur.

Die untere Auflösungsgrenze der Zeit in einem High-Level-Modul sind die Taktzyklen, die in den `always`-Blöcken des Moduls zur Synchronisation von Anweisungen verwendet werden. Mit den bei Zuweisungen an Ausgabeports möglichen Verzögerungen sind Laufzeiten nur andeutungsweise darstellbar. Ihr Effekt ist lediglich außerhalb des Moduls sichtbar und beschreibt ausschließlich die Ausgabeverzögerung der den Ports zugeordneten Register (*propagation delay*), wodurch sie im Gegensatz zu RTL-Modellen auf die logische Funktion des Moduls keine Auswirkungen haben können.

Eine präzise Zuordnung von Anweisungen und Operationen zu Takten ist nur durch spezielle Kombinationen von Modellierung und Optimierung erreichbar, bei denen High-Level-Optimierungen weitgehend ausgeschlossen werden. Für die allgemeine Anwendung der High-Level-Synthese bietet eine gröbere Betrachtung der Zeit in Form von Zeitschritten einen größeren Optimierungs- und Ergebnis-

spielraum. Ein Zeitschritt ist im Ablauf der Anweisungen eines `always`-Blockes die Zeitspanne zwischen zwei @-Anweisungen und besteht je nach darin enthaltenen Anweisungen, gewählten Optimierungen und geforderten Zielvorgaben aus einem oder mehreren Takten. Dabei sind zwischen verschiedenen Zeitschritten Unterschiede in der Taktzahl möglich, wie die Darstellung in Bild 4.2 erkennen läßt.

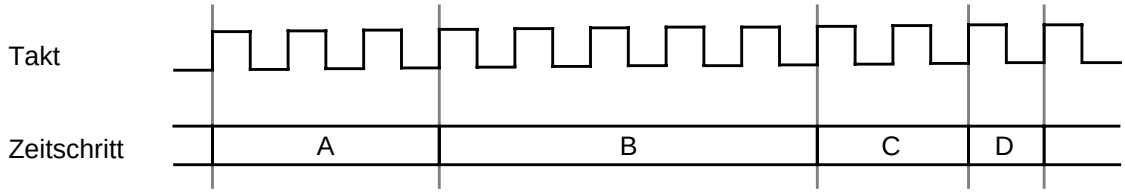


Bild 4.2: Verschiedene Zuordnungen von Zeitschritten zu Takten

Die Beschreibung struktureller Eigenschaften des High-Level-Moduls erfolgt durch die genannten Modellierungskonstrukte, wobei Continuous-Assignments, Ports und Instanzen wie bei der RTL-Synthese verwendet werden. Die `always`-Blöcke des Moduls werden auf komplexe Komponenten abgebildet, die sich aus einem Datenpfad und einem endlichen Automaten zusammensetzen und auf die algorithmische Beschreibung eines Blockes abgestimmt sind. Das Grundprinzip dieser komplexen Komponenten ist die FSMD-Architektur (*finite state machine with datapath*), eine zur Umsetzung beliebiger algorithmischer Beschreibungen in getaktete Logik geeignete Hardwarestruktur [Gajski97] [HsuTsa95]. Durch den Behavior-Compiler wird diese Grundkonstruktion gegebenenfalls durch Speicherfelder und Interfaces zu deren Ansteuerung erweitert, wie Bild 4.3 zeigt.

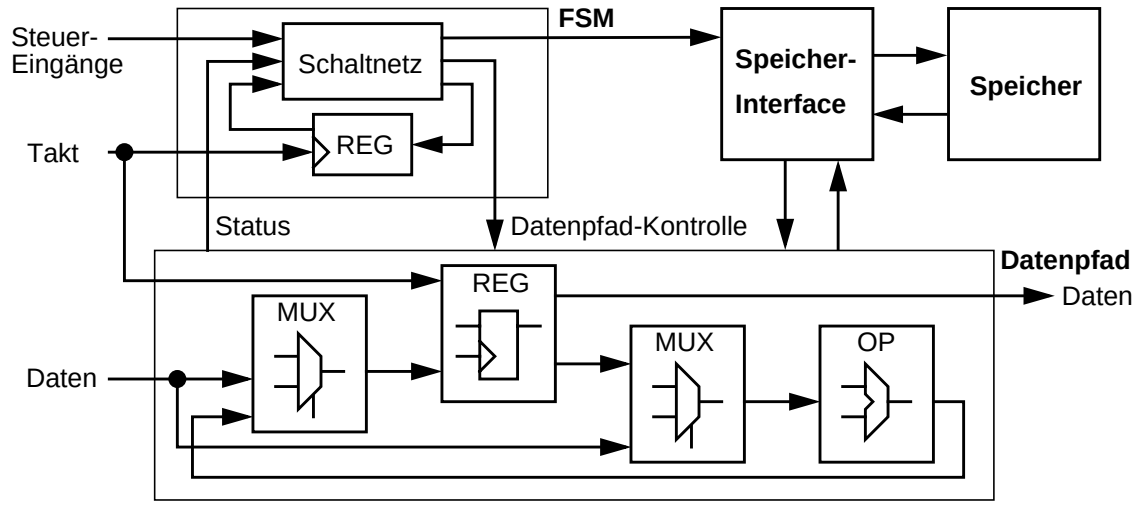


Bild 4.3: FSMD-Architektur des Behavior-Compilers

Die Modellierung eines `always`-Blockes hat nur in beschränktem Maß Einfluß auf die Struktur seiner FSMD-Umsetzung. Die benutzten Taktsynchronisationen und Operationen wirken sich auf die Anzahl und Verschaltung der Komponenten des Datenpfades und deren Ansteuerung durch die FSM aus, wobei die endgültige

Festlegung durch die Optimierung und deren Optionen erfolgt. Aus der Sicht des Entwicklers stellt bei der High-Level-Synthese ein `always`-Block durch die ihm zugeordnete FSMD eine atomare Entwurfskomponente dar.

Bei der Hierarchieebene der Entwurfsbeschreibung und der Verarbeitung von Daten deckt die High-Level-Synthese das gesamte Spektrum von Logik-, RT- und Algorithmusebene ab, wobei Datenmanipulationen durch logische Verknüpfungen und algorithmische Beschreibungen erfolgen.

4.2 Arbeitsschritte der High-Level-Synthese

Der Ablauf der High-Level-Synthese gliedert sich in mehrere Arbeitsschritte auf, die in festgelegter Reihenfolge aufeinander aufbauen und durch den Entwickler in unterschiedlichem Umfang gesteuert werden können (Bild 4.4). Diese Schritte werden in den nächsten Unterabschnitten näher betrachtet.

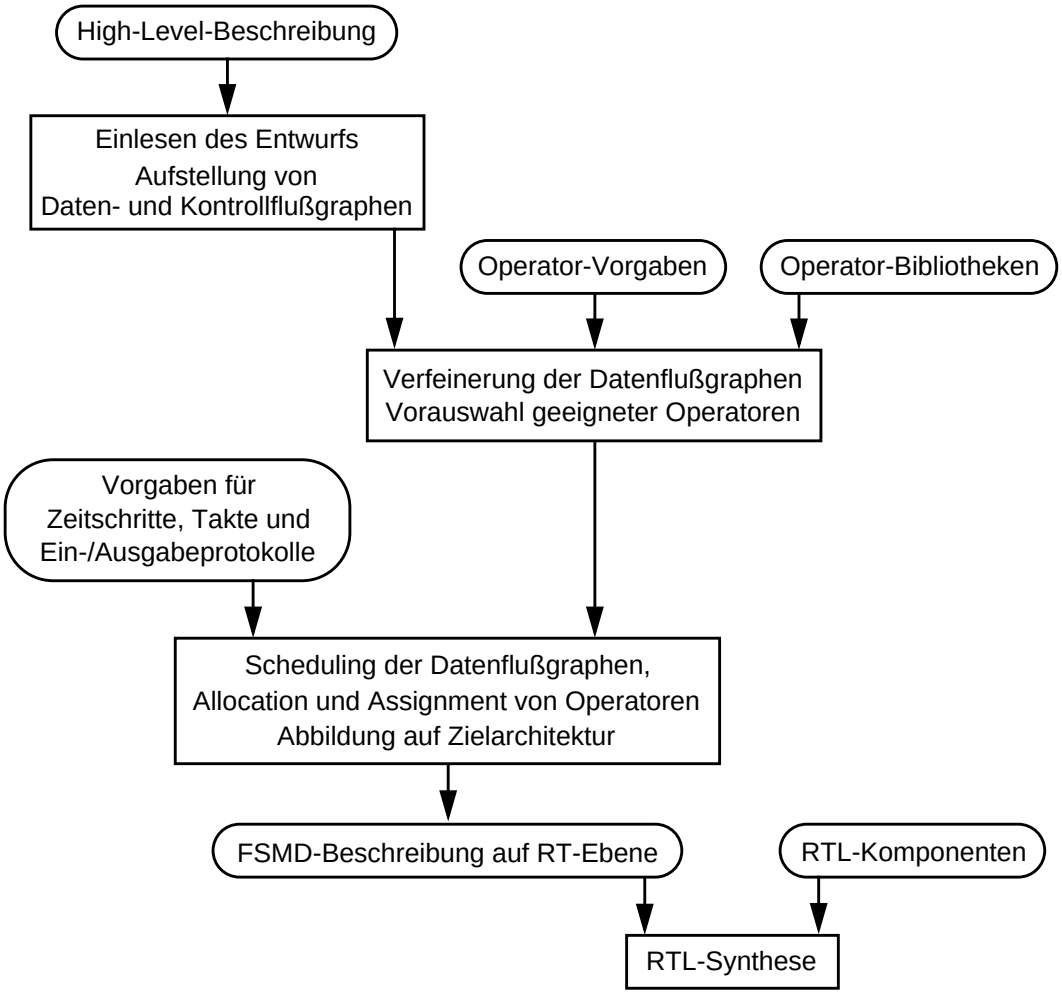


Bild 4.4: Arbeitsschritte der High-Level-Synthese

4.2.1 Aufstellung von Daten- und Kontrollflußgraphen

Nach dem Einlesen einer High-Level-Beschreibung und ihrer damit verbundenen syntaktischen und semantischen Prüfung durch das Synthesesystem stellt dieses

neben den Grobstruktur-Netzlisten für die RTL-Komponenten Datenflußgraphen für die algorithmischen Zuweisungsabläufe auf. In diesen gerichteten Graphen sind Verknüpfungsoperationen von Daten und ihre Abhängigkeiten verzeichnet. Außerdem werden Kontrollflußgraphen bestimmt, welche die Zusammenhänge zwischen mehreren Datenflußgraphen eines `always`-Blockes aufgrund bedingter Ausführungen und Schleifen erfassen.

Beispiel 4.1 Die Berechnung von $Y = (A + B + C) \cdot D$ in einer algorithmischen Beschreibung führt im ersten Schritt der High-Level-Synthese zur Aufstellung des Datenflußgraphen (DFG) in Bild 4.5. Die Operationen Lesen, Schreiben, Addieren und Multiplizieren sind die Knoten dieses Graphen, deren Datenabhängigkeiten durch die gerichteten Kanten angegeben sind.

```
module beispiel (Y, A, B, C, D, CLK);
  output [7:0] Y;
  input  [7:0] A, B, C, D;
  input        CLK;

  reg    [7:0] Y;

  always begin
    @(posedge CLK);
    Y = ( A + B + C ) * D;
  end

endmodule
```

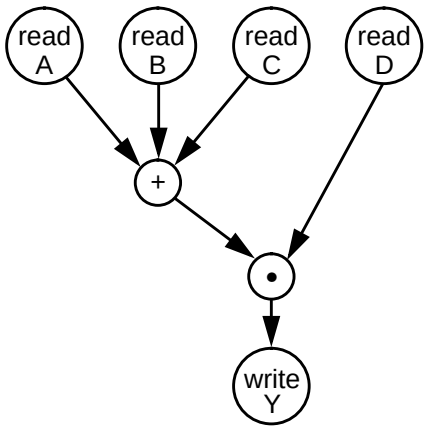


Bild 4.5: Einfache High-Level-Beschreibung und ihr Datenflußgraph

Die für jeden zusammenhängenden Zuweisungsablauf separat bestimmten Datenflußgraphen werden im nächsten Schritt weiter verfeinert, indem den Operationen gemäß der gewählten Optimierungsziele und der verfügbaren Hardware geeignete Baugruppen zugeordnet werden. Dabei handelt es sich zunächst nur um eine Vorauswahl, die noch alle realisierbaren Möglichkeiten offenläßt. Kriterien für die Auswahl geeigneter Baugruppen sind in diesem Schritt:

- die Anzahl der parallel verarbeiteten Operanden
- die Bitbreite der Operanden
- der explizite Ausschluß einer Baugruppe als Optimierungsziel
- die explizite Auswahl einer Baugruppe als Optimierungsziel

Diese Vorauswahl ermöglicht eine erste Abschätzung der Verarbeitungszeit und benutzbarer Komponenten. Sie liefert eine wichtige Entscheidungsgrundlage für den Entwickler bei der Auswahl der Optimierungsziele für die nachfolgenden Syntheseschritte und wird beim Behavior-Compiler als einzelner Arbeitsschritt mit `bc_time_design` gestartet [SynBCU97].

Beispiel 4.2 Der Datenflußgraph aus Beispiel 4.1 enthält neben Ports für je einen 8 Bit Operanden eine Multiplikation für zwei 8 Bit Operanden und eine Addition über drei 8 Bit Operanden. Ist in den verwendeten Hardwarebibliotheken jedoch

kein Addierer für drei Werte vorhanden, so ist ähnlich Bild 4.6 eine Aufteilung in einfachere Komponenten nötig. Die genaue Art der Aufteilung richtet sich nach den vorgegebenen Optimierungszielen und Umgebungsbedingungen.

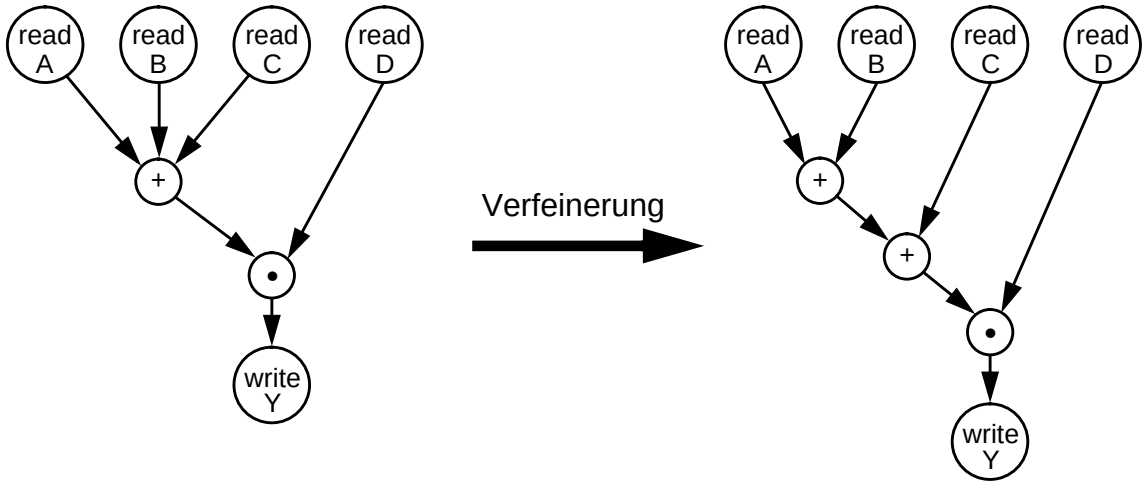


Bild 4.6: Aufteilung eines Additionsoperators im Datenflußgraphen

4.2.2 Scheduling, Operator-Allocation und -Assignment

Die im vorangegangen Schritt verfeinerten Datenflußgraphen werden als nächstes vom Entwickler mit Vorgaben über die maximale Durchlaufszeit der Graphen in Taktzyklen versehen. Zudem werden die Takte in ihrer Periodendauer spezifiziert, so daß für die endgültige Auswahl geeigneter Hardwareoperatoren detaillierte zeitliche Angaben vorliegen.

Auf dieser Grundlage werden vom Synthesesystem die Operationen eines DFG zunächst über die in der Beschreibung vorgegebenen Taktsynchronisationen hinaus in Zeitschritte unterteilt (*Scheduling*). Eng verzahnt mit der Aufstellung des zeitlichen Ablaufplanes (*Schedule*) ist die Bestimmung der benötigten Hardwareoperatoren (*Allocation*) und deren Zuordnung zu den Operationen des DFG (*Assignment*), wobei je nach Schedule ein Operator zu verschiedenen Zeiten gleichartige Operationen an mehreren Positionen des DFG bearbeiten kann [Gerez99]. Die Zeitschritte werden entsprechend der darin verwendeten Operatoren und der Zeitvorgaben weiter in Takte unterteilt. Das Ziel dieses Syntheseschrittes ist beim Behavior-Compiler die Erfüllung der gegebenen Zeitvorgaben mit möglichst wenig Hardware (*time-constrained synthesis*) [SynBCM97].

Beispiel 4.3 Der Datenflußgraph aus Beispiel 4.2 bietet mehrere Möglichkeiten zur Unterteilung in die Zeitschritte eines Schedules. In Bild 4.7 sind zwei dieser Schedules einander gegenübergestellt, die unterschiedliche Operator-Allocationen und -Assignments ermöglichen. Der Schedule in der linken Bildhälfte erlaubt nur beim Leseport der Variablen D die gemeinsame Nutzung mit einem anderen Port und erfordert für alle anderen DFG-Operationen eine eigene Hardwareressource. Dafür bietet er aber die kürzeste Ausführungszeit in Zeitschritten und Takten. Der Schedule in der rechten Bildhälfte benötigt lediglich einen Hardwareoperator für jeden Operationstyp und nutzt diesen jeweils mehrfach aus. Die Ersparnis an

Hardware wird dabei durch eine größere Ausführungszeit in Zeitschritten und Takten erkauft.

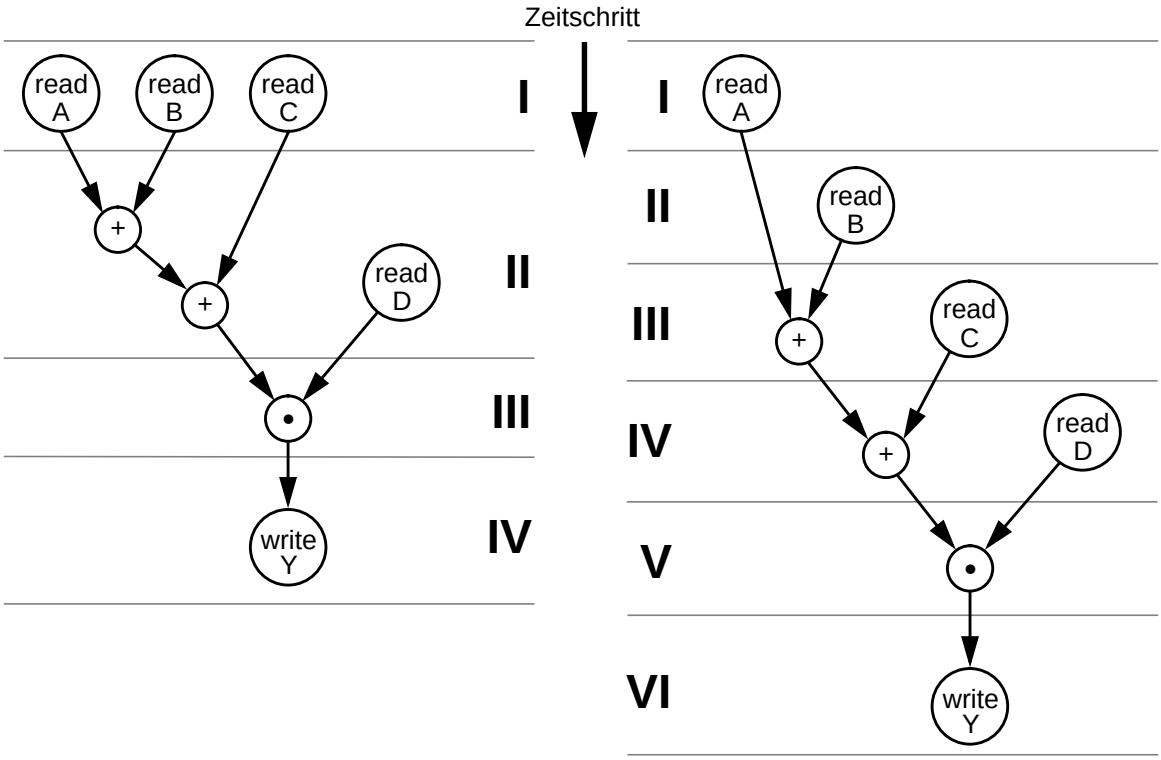


Bild 4.7: Verschiedene Schedules eines Datenflußgraphen

Beim Scheduling können neben zeitlichen Rahmenbedingungen auch verschiedene Modelle für das Ein-/Ausgabeprotokoll vorgegeben werden. So sind beim Behavior-Compiler drei Verfahren für Ein- und Ausgaben wählbar:

1. Taktfixierte Ein- und Ausgaben (cycle fixed mode)

   Bei diesem Ein-/Ausgabeprotokoll werden die Taktsynchronisationen direkt als Grenzen für Zeitschritte verwendet und jedem Zeitschritt nur ein Takt zugeordnet. Die Kontrolle über den benutzten Schedule und die Zuordnung von Operationen zu Takten liegt somit vollständig beim Entwickler, wodurch keine automatischen High-Level-Optimierungen beim Scheduling möglich sind. Bei Allocation und Assignment sind die Optimierungen auf die Auswahl und Festlegung von Operatoren zur Erfüllung der Taktvorgaben beschränkt.

2. Zeitschrittgebundene Ein- und Ausgaben (superstate mode)

   Hierbei werden die Ein-/Ausgabeoperationen durch Taktsynchronisationen des Modells verschiedenen Zeitschritten zugeordnet, wobei beim Scheduling weitere Zeitschritte eingefügt werden können. Die Dauer eines Zeitschrittes kann mehrere Takte betragen. Der Entwickler ist hierdurch in der Lage, die Ein-/Ausgabeoperationen gezielt relativ zueinander zu positionieren, ohne auf High-Level-Optimierungen bei Scheduling, Allocation und Assignment zu verzichten.

3. Freie Ein-/Ausgabeoperationen (free floating mode)

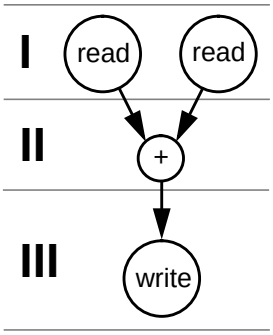
In diesem Ein-/Ausgabemodell kann der Entwickler durch die Abfolge von Zuweisungs- und Leseoperationen und deren Datenabhängigkeiten die Ein-/Ausgaben nur in grobem Rahmen relativ zueinander positionieren. Die Zuordnung der Ein-/Ausgaben zu Zeitschritten und deren Taktaufteilung werden vollständig durch das Scheduling bestimmt, so daß die High-Level-Optimierungen maximale Freiheiten besitzen.

Eine weitere Scheduling-Optimierung ist für Schleifen möglich, zwischen deren Iterationen keine Datenabhängigkeiten bestehen. In diesem Fall sind Schedules konstruierbar, bei denen sich die iterativen Ausführungen des Schleifenrumpfes zeitlich überlappen (*pipelined loop schedule*). Dabei werden die Operatoren einer bereits abgeschlossenen Anweisung noch vor Ende einer Schleifeniteration für den nächsten Schleifendurchlauf benutzt.

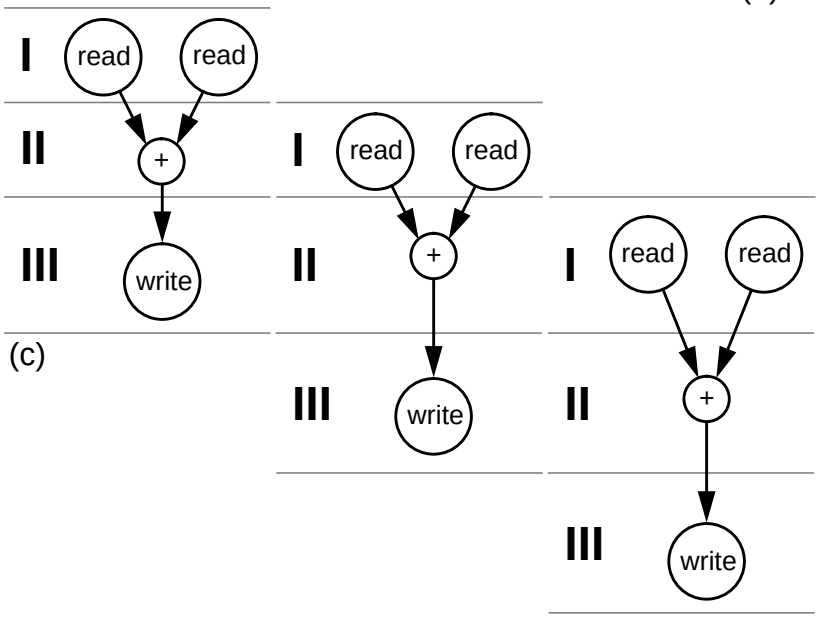
Beispiel 4.4 Die Summationsschleife in Bild 4.8 (a) addiert über drei Iterationen Werte der beiden Eingangsdaten-Arrays zu einem Ergebnis-Array auf, wobei keine Datenabhängigkeiten zwischen den einzelnen Läufen bestehen. Wird die Schleife in drei Zeitschritte für Lesen, Addieren und Schreiben unterteilt (b), so kann das Lesen für die zweite Iteration parallel zur Addition der ersten Iteration stattfinden und das Lesen der dritten Iteration parallel zur Addition der zweiten bzw. zum Schreiben der ersten Iteration (c).

```
reg [7:0] IN1[1:3], IN2[1:3], OUT[1:3];
integer   IDX;

always begin
  for (IDX = 1 ; IDX <= 3; IDX = IDX + 1) begin
    @(posedge CLOCK);
    OUT[IDX] = IN1[IDX] + IN2[IDX];
  end
end
```

(a)



(b)



(c)

Bild 4.8: Pipeline-Scheduling einer Schleife

Der Vorteil eines Pipeline-Schedule gegenüber einem rein sequentiellen Schedule liegt in der insgesamt kürzeren Ausführungszeit der Schleife und deren höherem Datendurchsatz. Es kann dabei aber zu einem im Vergleich höheren Bedarf an Hardwareressourcen kommen, so daß dieses Scheduling nicht unbedingt in jeder Hinsicht die besten Lösungen liefert.

Aus Benutzersicht parallel zu Scheduling, Allocation und Assignment werden die Variablen der High-Level-Beschreibung auf ihre Verwendung und die Dauer ihrer Belegung mit Werten (Lebenszeit) hin untersucht. Bei der Zuordnung von Variablen auf Hardwareregister werden dann Variablen mit nichtüberlappenden Lebenszeiten auf gemeinsam genutzte Register abgebildet. Register sind damit ebenso wie die Operatoren von der High-Level-Synthese verwaltete und optimiert zugewiesene Ressourcen.

Bei der Optimierung der Operatornutzung und der Registerbelegung wirken sich unabhängig vom benutzten Ein-/Ausgabeprotokoll Taktsynchronisationen als Einschränkungen aus. Sie führen zu einer Zuordnung von Rechenoperationen und Registerbelegungen in getrennte Zeitschritte und verhindern damit die freie Belegung von Zeitschritten mit Operationen. Dies kann bei kritischen Vorgaben für die Anzahl der Takte und die Taktperiodenlänge zu einem hohen Bedarf an Ressourcen führen oder gar die Aufstellung eines Schedule verhindern, so daß durch die High-Level-Synthese kein Ergebnis geliefert werden kann. Die Nutzung von Taktsynchronisationen sollte daher vom Entwickler möglichst vermieden werden, um den Optimierungsspielraum nicht unnötig zu verkleinern.

4.2.3 Abbildung auf die Zielarchitektur

Mit der Abbildung der Kontrollflußgraphen sowie der mit Scheduling, Operator-Allocation und -Assignment behandelten Datenflußgraphen auf die bereits in Bild 4.3 vorgestellte FSMD-Architektur entsteht eine RTL-Entwurfsbeschreibung. Es sind dabei jedoch einige Besonderheiten zu beachten, die sich rückwirkend auf die Möglichkeiten des Scheduling und das von der Schaltung einforderbare Timing auswirken.

Die Basisstruktur des Datenpfades, die in Bild 4.9 detailliert dargestellt ist, führt zu mehreren zeitlichen Mindesterfordernissen für Berechnungsabläufe und die Synchronisation mit der steuernden FSM.

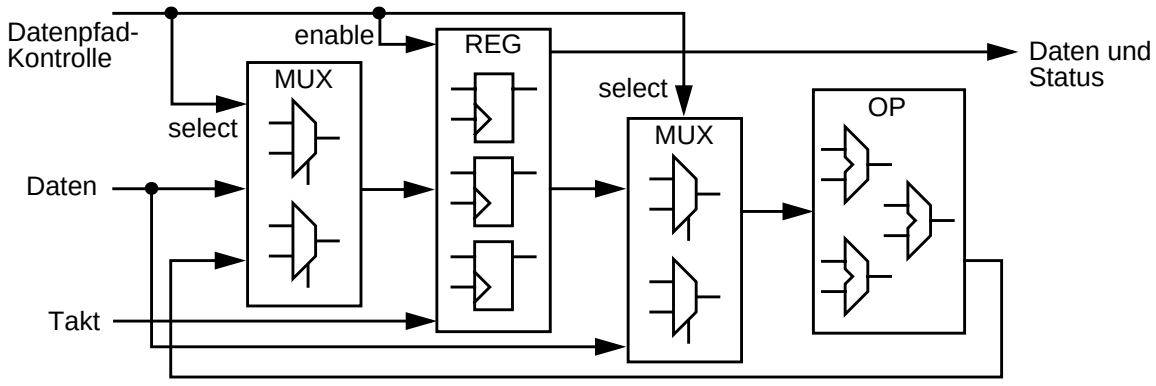


Bild 4.9: Struktur des FSMD-Datenpfades

So werden grundsätzlich alle Ausgänge des Datenpfades von der Registerbank (REG) angesteuert. Dies führt zu einer Reaktionszeit des Datenpfades bezüglich berechneter Daten und dem Status gegenüber Eingangsdaten und Steuersignalen von einem Takt. Bei aufeinanderfolgenden Berechnungen, bei denen der an die FSM übermittelte Status über den weiteren Ablauf entscheidet, bildet die Anzahl dieser Berechnungen das Minimum der für das Scheduling nötigen Zeitschritte. Ist beispielsweise bei einer `if`-Anweisung die Ausführung einer Berechnung von einem anderen Berechnungsergebnis abhängig, so ist im ersten Zeitschritt die Ausführungsbedingung zu berechnen, bevor in einem weiteren Zeitschritt die bedingte Berechnung erfolgen kann. Hierdurch erklärt sich der Mindestbedarf von zwei Zeitschritten für bedingte Ausführungen, der für ein erfolgreiches Scheduling nicht durch niedrigere Vorgaben unterboten werden kann.

In dem Operator-Block (OP) können die Hardwareoperatoren abhängig von den Taktvorgaben für die Berechnungsdauer einzeln ansteuerbar vorliegen oder zu Operatornetzen kaskadiert sein. Die einzelne Ansteuerung ermöglicht eine mehrfache Nutzung von Operatoren, die sich aber jeweils über einen Zeitschritt erstrecken, so daß zur Nutzung dieser Implementierung hohe Ausführungszeiten zur Verfügung stehen müssen. Zu Operatornetzen kaskadierte Operatoren sind nur beschränkt auf ihre Gesamtberechnung mehrfach verwendbar, die aber dafür in einem Zeitschritt ausführbar ist. Die Zeitvorgaben für das Scheduling haben im Operator-Bereich des Datenpfades also signifikante Auswirkungen auf die Menge, die Anordnung und den Zeitverbrauch der Operatoren.

Der den Datenpfad steuernde endliche Automat, der in Bild 4.10 dargestellt ist, besteht aus einem Schaltnetz und einem Zustandsregister, die in Form eines Mealy-Automaten miteinander verbunden sind. Damit sind Ausgangsreaktionen der FSM asynchron infolge von Statusänderungen des Datenpfades möglich. Zur Verkürzung von Signalpfaden bei knapp bemessener Taktperiode können dem Schaltnetz jedoch Register vor- und nachgestellt werden, so daß es zu Reaktionsverzögerungen von bis zu zwei Takten kommen kann.

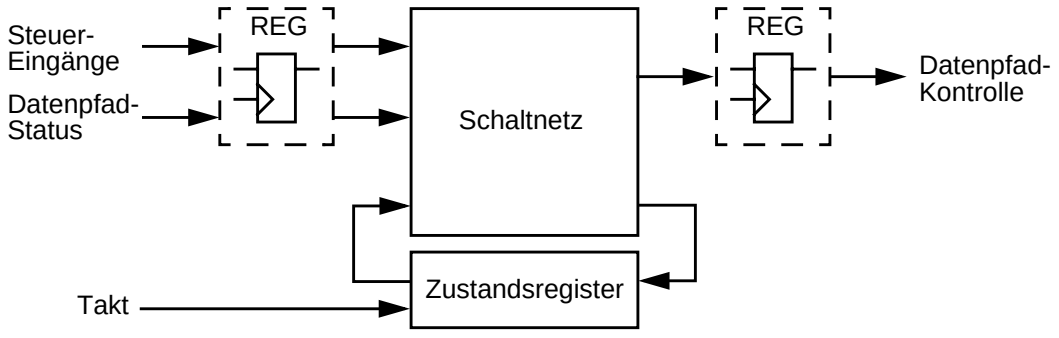


Bild 4.10: Struktur des FSMD-Steuerautomaten

Bei der Abbildung des Kontrollflusses eines `always`-Blockes auf den Automaten werden für den Blockrumpf wie bei dem Schleifenrumpf einer `for`- oder `while`-Schleife ein Startzustand und von diesem getrennte Ausstiegs- und Fortsetzungszustände reserviert. Ein `always`-Block erfordert daher immer mindestens zwei

Zustände bzw. Takte. Durch bedingte Berechnungen und Taktsynchronisationen erhöht sich diese Anzahl weiter. Die sich hieraus ergebende Mindestanzahl von Takten stellt die Untergrenze der für ein erfolgreiches Scheduling möglichen Taktzahlvorgabe dar. Abhängig von den Zeitvorgaben kann der Automat darüber hinaus weitere Zustände zur Ablaufsteuerung der Operations-Schedules und der Ein-/Ausgabeprotokolle erhalten.

4.3 Optimierungsmöglichkeiten der High-Level-Synthese

Die in den vorangegangenen Abschnitten im Rahmen des Syntheseablaufes kurz vorgestellten Optimierungsmöglichkeiten der High-Level-Synthese werden hier noch einmal zusammenfassend aufgeführt. Zu ihnen gehören:

- Die automatische Erstellung eines ressourcenfreundlichen Ablaufplanes für die Ausführung von Operationen (Schedule), wobei auch eine überlappende Ausführung von Schleifen oder spekulative Berechnungen mit ansonsten freien Operatoren angewendet werden.
- Die Auswahl der günstigsten Operatoren für die geforderten Zeitvorgaben (Allocation) und eine optimierte Zuordnung von Operationen zu Operatoren (Assignment) mit möglichst mehrfacher Ressourcennutzung.
- Eine optimierte Zuordnung von Variablen zu Hardwareregistern auf der Grundlage einer Belegungsanalyse.

Diese Optimierungen können durch Taktsynchronisationen der Modellierung und durch kaum erreichbare Zeitvorgaben in ihren Möglichkeiten stark eingeschränkt werden. Es liegt also weitgehend in der Hand des Entwicklers, das Potential dieser Optimierungen durch Freiheiten in der Verarbeitungszeit nutzbar zu machen.

Die Optimierung von Entwurfselementen höherer Abstraktionsebenen, beispielsweise von Algorithmen sowie deren Daten- und Kontrollfluß, liegen beim Behavior-Compiler noch vollständig auf der Seite des Entwicklers. Es gibt somit bei der hier betrachteten Form der High-Level-Synthese noch ein großes Potential an Optimierungsmöglichkeiten der Modellierung außerhalb der Zeitvorgaben.

5 Controllersynthese

Anhand des Protocol-Compilers wird in diesem Kapitel die Controllersynthese vorgestellt. Obwohl der Protocol-Compiler und die mit ihm erzeugbaren Controller besonders für die Bearbeitung von Protokollen der Datenkommunikation geeignet sind, gelten viele der betrachteten Dinge ebenso für andere Werkzeuge des Controllerentwurfs und ihre Ergebnisse.

Die zur Eingabe der Entwürfe verwendete Protocol-Compiler-HDL und ihre Eigenschaften wurden bereits in den Abschnitten 2.2.1 und 2.3.2 dieser Arbeit behandelt und mit dem standardisierten Verilog-Sprachumfang verglichen. Die Protocol-Compiler-HDL wird daher hier nicht weiter betrachtet. Der Schwerpunkt dieses Kapitels liegt bei den Arbeitsschritten der Controllersynthese [SynPCU99] auf dem Weg zur RTL-Synthese mit dem Design-Compiler.

Neben den Attributen eines Entwurfs und seiner Elemente für die Synthesesteuerung werden dabei auch die bei der Synthese ausführbaren Optimierungen und ihr Einfluß auf die Hardwareumsetzung erläutert. Der Aufbau dieses Kapitels orientiert sich an der in Bild 5.1 dargestellten Abfolge von Entwurfsschritten mit dem Protocol-Compiler, die zum Erreichen einer Controllerbeschreibung für die RTL-Synthese benötigt werden. Zur Vereinfachung der Darstellung werden die in dieser Abfolge vorkommenden Verifikationen nur als Nebenzweige angedeutet und nicht weiter ausgeführt.

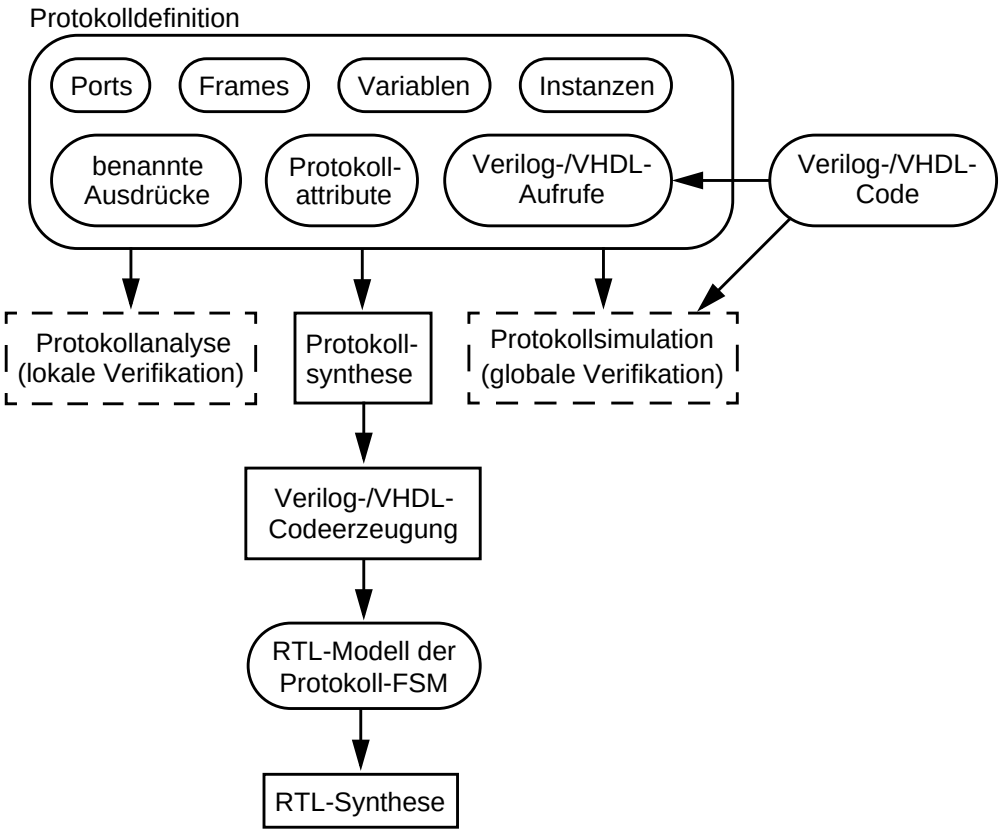


Bild 5.1: Entwurfsschritte der Controllersynthese

In diesem Ablauf erfolgt zunächst die Beschreibung des Controllerentwurfes in der integrierten Entwicklungsumgebung des Protocol-Compilers. Der Entwurf ist dabei in mehrere Teilbereiche untergliedert, die im Zusammenspiel miteinander die vollständige Definition der Protokollfunktion des Controllers ergeben. Neben der Definition des Protokolls kann in diesen Entwurfsteilen durch Attribute die Implementierung des Controllers gesteuert werden, ohne dabei seine Funktion zu beeinflussen. Die Protokollteile und ihre Attribute werden ebenso wie die Möglichkeiten und Auswirkungen der Protokollsynthese und der Codeerzeugung auf die Controllerstruktur in den folgenden Abschnitten betrachtet.

5.1 Teile einer Protokolldefinition

Die Teilbereiche der Protokolldefinition beschreiben unterschiedliche Elemente des Controllers und ihre Eigenschaften. Sie werden in diesem Abschnitt kurz vorgestellt.

Ports: Die Signalschnittstellen des Controllers zu den umgebenden Schaltungen können ein oder mehr Bit Breite besitzen und entweder als Eingabe- oder als Ausgabeports spezifiziert werden. Die Ausgabeports sind tri-state-fähig, in ihrem Anfangswert bestimmbar und über Ausgaberegister mit dem Takt des Controllers synchronisierbar. Zwei Eingangsports sind für jeden Entwurf vorab definiert, ein Takteingang für die Zustandsfortschaltung und ein Reset-Eingang zur Initialisierung. Beim Takt kann entweder die steigende oder die fallende Signalflanke als aktiver Zeitpunkt gewählt werden. Für den Reset-Eingang sind die aktive Signalpolarität und der Zeitpunkt der Verarbeitung bestimmbar, wobei letzterer synchron zum Takt oder asynchron bei Erreichen des aktiven Pegels liegt. Alle Ports sind global im Controllerentwurf sichtbar und benutzbar.

Frames: Die Beschreibung der Controllerfunktion erfolgt im wesentlichen mit der Protocol-Compiler-HDL innerhalb und durch Frames und den an sie gebundenen Actions, die mit Operatoren ähnlich Produktionen formaler Sprachen zusammengefügt werden (Abschnitt 2.3.2). Die Umsetzung dieser Beschreibung in einen endlichen Automaten durch die Protokollsynthese kann mit mehreren Attributen gesteuert werden, welche die Controllerfunktion selbst nicht beeinflussen.

Zum einen ist der Controller in seinem Zustandsraum partitionierbar, wobei die Partitionsgrenzen der Aufteilung in Frames folgen, indem die Codierung der Zustände von Frame-Hierarchien durch die Protokollsynthese über ein Attribut vorgegeben wird. Zur Auswahl stehen neben dem standardmäßig verwendeten und sich über den gesamten Controller verteilenden Aktivitätscode der Frames (*distributed encoded*) Codes für eine optimierte Anzahl von Zustandsbits (*binary min-encoded*), geringe Codewortdistanzen (*min-distance min-encoded*), am Ablauf der Frames orientierte Zustandsfolgen (*branch-sequence encoded*) und 1-aus-n-Zustandsvektoren (*one hot encoded*).

Zum anderen kann die Implementierung bei Wiederholungszählern gesteuert werden. Neben der Auflösung eines Zählers in der Zustandsmenge seiner Partition und ihrer Codierung ist der hierarchisch getrennte Aufbau als Binärzähler oder *linear feedback shift register* (LFSR) möglich.

Die Steuerung von Pipeline-Verarbeitungen, die signifikanten Einfluß auf die Größe und die Komplexität der Schaltung hat, wirkt sich auf die Funktion zwar nur vergleichsweise unscheinbar aus, ist aber trotzdem kein reines Implementierungsattribut. Durch sie kann einem sequentiellen Frame-Ablauf ermöglicht werden, an mehreren Positionen parallel aktiv zu arbeiten. Hierzu müssen nicht nur einzelne Frames in Zustände umgesetzt werden, wie bei der auf ein aktives Frame beschränkten Verarbeitung, sondern alle Kombinationsmöglichkeiten für aktive Frames.

Variablen: Die Speicherung von Daten im Controller erfolgt in einzelnen Registern mit ein oder mehr Bit Breite, die als Teil der Protokolldefinition global benutzbar sind. Schreibzugriffe werden parallel zur Zustandsfortschaltung bei den aktiven Taktflanken ausgeführt. Bei der Deklaration eines Variablenregisters ist optional ein Initialisierungswert definierbar, den die Variable bei aktivem Reset erhält, sowie ein Standardwert (*default value*), der immer dann eingenommen wird, wenn kein expliziter Schreibzugriff auf die Variable erfolgt. Mit Standardwerten lassen sich insbesondere Kommunikationssignale mit Impulscharakter gut darstellen, da nur der Pulsbeginn explizit modelliert werden muß, während die Rückstellung in der Variable fest verankert ist.

Benannte Ausdrücke: Mehrfach im Protokoll benötigte Berechnungen einzelner Ausdrücke sind getrennt von Frames und ihren Aktionen definierbar. Die Ausdrücke sind in ihren Referenzen auf Ports, Variablen, Konstanten und Verilog- bzw. VHDL-Funktionen festgelegt und somit frei von Parametern. Ein benannter Ausdruck kann in Frames, Actions oder Ausdrücken wie ein Port oder eine Variable lesend referenziert werden.

Instanzen: Die Einbindung von RTL-Entwürfen der Sprachen Verilog oder VHDL in den Controller, die in sich abgeschlossen und für eine RTL-Synthese geeignet sind, erfolgt durch Instanzierung ihrer RTL-Module bzw. -Architekturen. Über die Ports oder Variablen des Controllerentwurfes erfolgt die Anbindung der Schnittstellensignale der Instanzen. Die eingebundenen RTL-Entwürfe müssen durchweg in der HDL beschrieben sein, die bei der Codeerzeugung für den Controller gewählt wird. Der innere Aufbau und die Funktion der Instanzen bzw. ihrer Schnittstellen wird vom Protocol-Compiler nicht betrachtet und nicht auf Konsistenz mit ihrer Deklaration für die Controllereinbindung überprüft. Die korrekte Einbindung der durch Attribute angegebenen Quelltextdateien ist vom Entwickler sicherzustellen.

HDL-Aufrufe: In Verilog oder VHDL beschriebene Berechnungen sind verpackt in Funktionen auch ohne umgebende Module bzw. Architekturen von der Controllerbeschreibung benutzbar. Sie werden bei der Codeerzeugung direkt in das RTL-Controllermodell eingefügt und sind mit ihm über die Aufrufparameter und ggf. den Rückgabewert verbunden. Sie müssen daher in der HDL vorliegen, die bei der Codeerzeugung für den Controller gewählt wird. Ebenso wie bei Instanzen muß die korrekte Einbindung auch hier vom Entwickler sichergestellt werden, da dem Protocol-Compiler nur die Aufrufdeklaration und der Name der einzufügenden Quelltextdatei bekannt sind.

Protokollattribute: Die Eigenschaften des gesamten Protokolls, die keinem der vorgenannten Bereiche eindeutig zugeordnet sind, werden mit globalen Attributen erfaßt, die sich übergreifend auf alle Teile der Protokolldefinition beziehen. Hierzu gehören neben den Informationen zur Wurzel der Entwurfshierarchie, der Simulatoranbindung und den verwendeten Dateibezeichnungen die Angaben zu Optimierungen der Protokollsynthese und der Codeerzeugung.

Für die Optimierungen bezüglich bestimmter Zustandscodierungen bestehen dieselben Möglichkeiten wie für einzelne Controllerpartitionen, wobei zusätzlich die den Zielvorgaben am besten entsprechende Codierung automatisch selektiert werden kann. Der für die Optimierungen zu investierende Aufwand ist einstellbar.

Bei der Codeerzeugung sind neben der Sprache für die synthesefähige RTL-Beschreibung und dem Hinzufügen von Verifikationshilfen (*debug code*) mehrere Controllerstrukturen wählbar. Auf diese wird im nächsten Abschnitt eingegangen.

5.2 Mögliche Controllerstrukturen

Bei der Codeerzeugung besteht die Auswahl zwischen drei Grundstrukturen für die Implementierung des Controllers. Alle drei sind jeweils in einem Verilog-Modul bzw. einer VHDL-Architektur für den Controller und die in ihn eingebundenen HDL-Codes realisiert. Ihre innere Aufteilung in einzelne Prozesse unterscheidet sie voneinander und bietet alternative Ansatzpunkte für die RTL-Synthese.

Single-Process-Struktur: Dies ist die einfachste der drei Controllerstrukturen, bei der die gesamte Protokollaktivität in einem getakteten Prozeß beschrieben wird. In diesem Prozeß erfolgen alle Zuweisungen an die Register des Controllers, wobei die Berechnung der im nächsten Takt aktiven Frames in eine untergeordnete Task (Verilog) bzw. Prozedur (VHDL) ausgelagert ist. Diese Trennung dient lediglich der organisatorischen Aufteilung von Zustands- und Datenpfadlogik und hat keine Auswirkungen auf die RTL-Synthese, bei der die Task bzw. Prozedur im Prozeß aufgelöst wird. Der Aufbau eines Controllers mit Single-Process-Struktur wird graphisch in Bild 5.2 verdeutlicht.

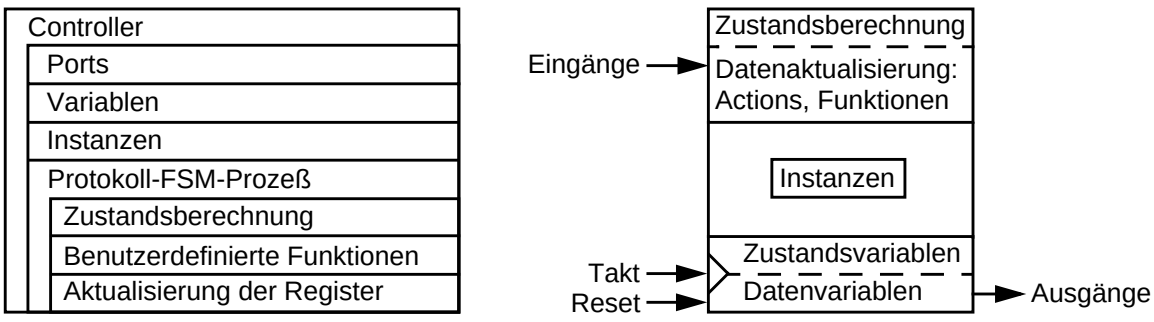


Bild 5.2: Aufbau eines Single-Process-Controllers

Split-Process-Struktur: In dieser Controllerstruktur werden zwei Prozesse für die Modellierung benutzt, von denen einer die kombinatorische Logik und der andere die getaktete Logik des Controllers darstellt. Bei der kombinatorischen Logik ist wie bei der Single-Process-Struktur die Berechnung der im nächsten Schritt

aktiven Frames ausgelagert. Durch die Aufteilung in einen rein kombinatorischen und einen vollständig getakteten Teil sind für die RTL-Synthese im Vergleich zur Single-Process-Struktur alternative Optimierungsansätze möglich, die zu anderen Ergebnissen führen können. Die Zusammensetzung der Split-Process-Struktur aus ihren Komponenten ist in Bild 5.3 dargestellt.

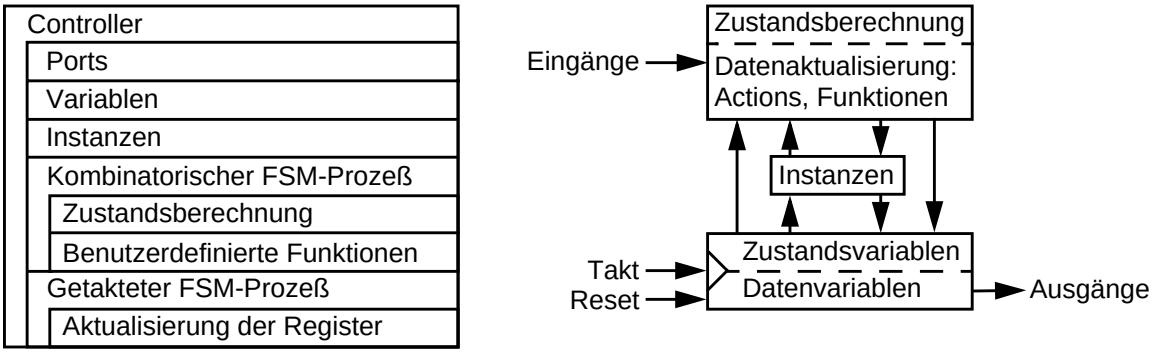


Bild 5.3: Aufbau eines Split-Process-Controllers

Multi-Process-Struktur: Diese Aufteilung des Controllers ist die detaillierteste der drei Arten und verwendet fünf parallele Prozesse zur Modellierung, unterteilt nach Protokollablaufsteuerung und Datenpfadlogik sowie kombinatorischer und getakteter Logik. Durch diese Unterteilung sind alternative Optimierungsansätze der RTL-Synthese im Vergleich zu den anderen zwei Controllerstrukturen möglich und eine weitere Vergrößerung des Ergebnisraumes erreichbar. Die Unterteilung des Controllers bei der Multi-Process-Struktur zeigt Bild 5.4.

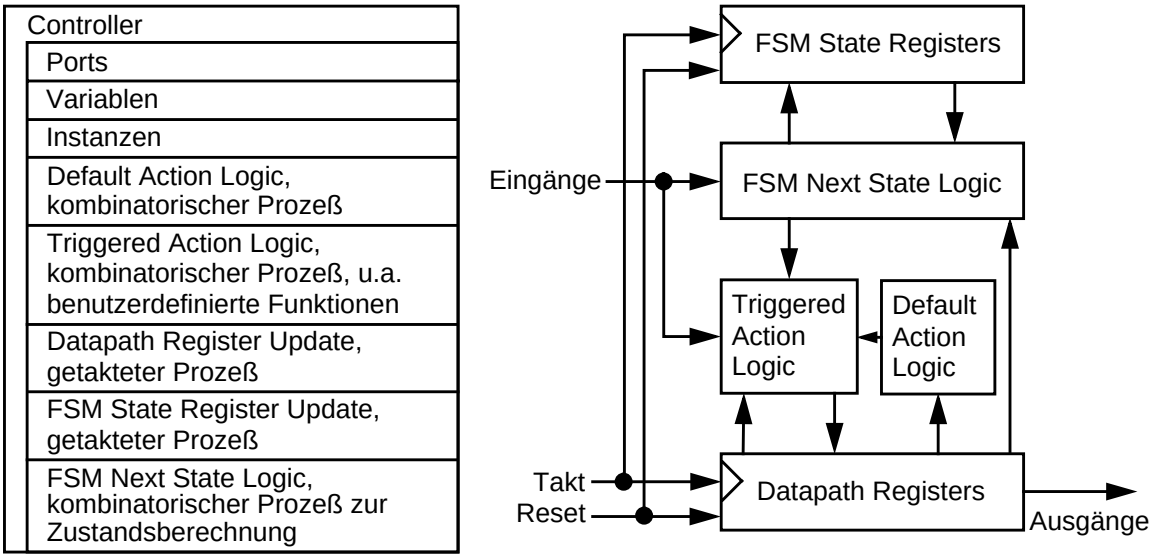


Bild 5.4: Aufbau eines Multi-Process-Controllers

5.3 Bezug zwischen Frame-Definitionen und FSM-Implementierung

Trotz der vielfachen Unterschiede zwischen einer Protokolldefinition und ihrer FSM-Implementierung in Darstellung und Struktur gibt es Elemente in beiden

Formen, die Bezugspunkte zwischen dem Modell und der Hardware herstellen. Die Kenntnis über diese Bezüge erlaubt bei der Modellierung die Nutzung bestimmter Implementierungsarten und deren Eigenschaften für einen Controller, wodurch eine insgesamt höhere Ergebnisqualität erzielbar ist.

Die Frames eines Modells und die Zustände des resultierenden Controllers bilden einen dieser Bezugspunkte, bei dem sowohl die Umsetzung von Frames in Zustände als auch der Aufbau verschiedener Zustandscodierungen interessant für den Entwickler sind. In besonderer Weise findet sich der Bezug zwischen Frames, Zuständen und Codierungen bei mehrfach ausgeführten Frames, die über Zähler gesteuert werden.

Ein weiterer Bezugspunkt ergibt sich durch die Umsetzung von Berechnungen innerhalb von Frame-Bedingungen, Actions und den darin aufgerufenen HDL-Codes in kombinatorische Logik. Da sich ihre Komplexität auf die Laufzeiten auswirkt und jeder Aufruf die Menge an benötigter Logik vergrößert, kann durch eine geeignete Aufteilung bzw. mehrfache Nutzung von Berechnungen das Timing bzw. die Logikmenge des Controllers bei der Modellierung erheblich beeinflußt werden.

In den folgenden Unterabschnitten werden die angesprochenen Bezugspunkte näher betrachtet, wobei mit dem allgemeinen Bezug zwischen Frames, Zuständen und der Codierung begonnen wird. Wiederholungszähler und kombinatorische Berechnungen werden in eigenen Unterabschnitten behandelt.

5.3.1 Frames, FSM-Zustände und Zustandscodierungen

Die Frames einer Controllerdefinition unterteilen sich, wie bereits in Abschnitt 2.3.2 vorgestellt, in primitive Frames und referenzierende Frames. Lediglich die primitiven Frames werden bei der Controllersynthese in FSM-Zustände überführt, während Referenz-Frames selbst nur Verweise auf Frame-Definitionen sind. Die zur Umsetzung einer Frame-Definition in einen Automaten benötigten Zustände sind von ihren Ausführungsmöglichkeiten abhängig, welche bei der Modellierung bestimmt und ggf. eingeschränkt werden können.

Im einfachsten Fall kann innerhalb einer Frame-Definition nur ein einziges primitives Frame aktiv sein, so daß im Rahmen dieser Definition jedes primitive Frame einem FSM-Zustand entspricht. Hierzu darf diese Frame-Definition weder alternative Ausführungspfade enthalten noch variable Wiederholungen, deren Abbruch nicht durch das nachfolgende Frame erzwungen wird.

Beispiel 5.1 Bei der Frame-Definition in Bild 5.5 wird auf dem Eingangsport IN in vier Takten nacheinander die Zeichenfolge „STOP“ abgefragt. In der Definition existieren keine Alternativen und keine Wiederholungen. Durch die sich gegenseitig ausschließenden Frame-Bedingungen ist zudem sichergestellt, daß bei diesem Frame in beliebigem Ausführungsumfeld nie mehr als ein primitives Frame aktiv sein kann. Daher wird bei der Controllersynthese jedem dieser Frames genau ein Zustand der Controller-FSM zugeordnet.

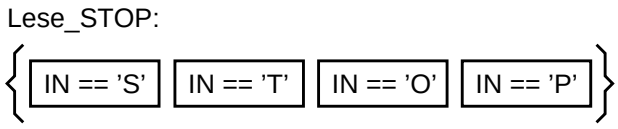


Bild 5.5: Frame-Definition mit direkter FSM-Zustandskorrespondenz

Durch den Alternativoperator und sich nicht gegenseitig ausschließende Frame-Bedingungen entstehen in einer Frame-Definition alternative Ausführungspfade. Die Zustände ergeben sich dann aus den einzeln aktiven Frames und den parallel aktiven Frame-Mengen.

Beispiel 5.2 Die Frame-Definition in Bild 5.6 prüft den Eingangsport IN auf die Zeichenfolgen „START“, „END“ und „STOP“ in drei alternativen Abfragepfaden. Da bei „START“ und „STOP“ die ersten beiden Zeichen identisch sind, werden die beiden Pfade in den ersten beiden Takten parallel ausgeführt. Andere Arten von paralleler Ausführung, wie z.B. Pipelining durch zeitverschobene Aktivierung des selben oder eines alternativen Pfades, sind aufgrund der Ablaufbedingungen nicht möglich. In der Controller-FSM belegen die Abfragen von „S“ und „T“ daher jeweils denselben Zustand, während den übrigen Frames jeweils ein eigener Zustand zugeordnet ist. Eine Frame-Definition mit nur einem Pfad für die Abfrage von „S“ und „T“ ließe dies direkt in der Definition erkennen, wäre aber nicht so änderungsfreundlich.

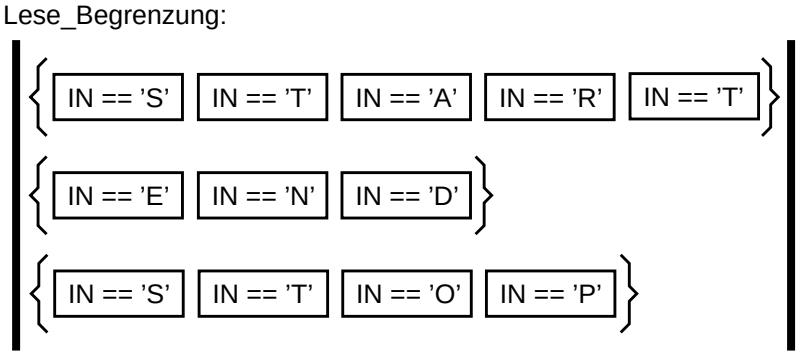
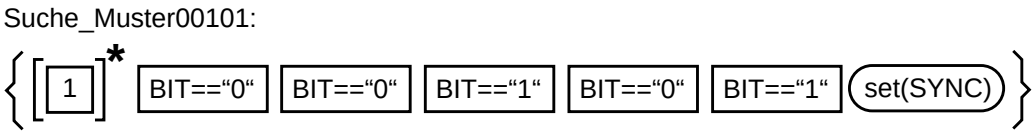


Bild 5.6: Alternative Ausführungspfade in einer Frame-Definition

Die parallele Ausführung primitiver Frames innerhalb eines Ausführungspfades mit Pipelining durch ihre zeitversetzte Aktivierung gehört von der Modellierung eng mit variablen Wiederholungen zusammen, die nicht mit der Aktivierung des nachfolgenden Frames beendet werden. Während derartige Wiederholungen als Auslöser für zeitversetzte Pfadaktivierungen fungieren, muß in der Pfadstruktur eine zeitversetzt parallele Ausführung mehrerer primitiver Frames durch sich nicht ausschließende Frame-Bedingungen ermöglicht werden. Ist ein Pipelining aufgrund des damit verbundenen Logikaufwandes trotz geeigneter Modellierung unerwünscht, so kann es durch ein Attribut unterdrückt werden.

Beispiel 5.3 Durch die Wiederholung des ersten Frames in Bild 5.7, die nicht mit der Aktivierung des nächsten Frames beendet wird, bleibt dieses dauerhaft aktiv und stößt mit jedem Takt den nachfolgenden Frame-Pfad an. Das in diesem Pfad ausgewertete Bitmuster erlaubt durch mehrere gleichartige Frame-Bedingungen den zeitversetzten Start des Ablaufes mit überlappender Ausführung. So wird bei der eintreffenden Bitfolge „00100101“ mit jedem 0-Bit eine Ausführung gestartet, die erst bei Abweichung des vorliegenden Musters von der Suchvorlage eingestellt wird. Aufgrund der Abhängigkeiten zwischen den Bedingungen, die nur wenige Kombinationsmöglichkeiten zulassen, ergeben sich insgesamt sechs Zustände und bis zu zwei Ausführungspfade. Angewendet wird eine solche Konstruktion meist zur Suche von Synchronisationsmustern in Datenströmen durch einen autonom arbeitenden Teilautomaten.



Takt	Bit	Aktivität
0	-	
1	0	
2	0	
3	1	
4	0	
5	0	
6	1	
7	0	
8	1	

Bild 5.7: Mustersuche mit Pipelining

Für die sich aus den Frame-Definitionen ergebenden Zustände eines Controllers gibt es mehrere Möglichkeiten zur Codierung, die sich auf Größe und Laufzeit der Zustandsverarbeitung auswirken und durch Attribute gesteuert werden können:

- Distributed Encoded
- Binary Min-Encoded
- Min-Distance Min-Encoded
- Branch-Sequence Encoded
- One Hot Encoded

Diese Codes werden im folgenden zunächst einzeln in Aufbau und Eigenschaften vorgestellt. Anhand eines einfachen Beispiels erfolgt danach ihr Vergleich.

Distributed Encoded: Bei dieser Codierung ist jedem Frame eines Protokolls ein 1-Bit Register zugeordnet, in welchem die Aktivität des Frames verzeichnet wird. Der Gesamtzustand des Controllers wird damit verteilt auf die einzelnen Frames codiert, wobei je nach Menge gleichzeitig aktiver Frames auch mehrere Register gesetzt sein können. Eine detaillierte Zustandsanalyse unter Zusammenfassung stets gleichzeitig aktiver Frames und Auslassung nicht erreichbarer Frames wird nur bei Auswahl des hohen Optimierungsaufwandes der Codierung durchgeführt, wofür die möglichen Frame-Aktivitäten untersucht werden. Es wird grundsätzlich eine große Menge an Zustandsregistern benötigt, die durch einfache und schnelle Zustandslogiken angesteuert werden.

Binary Min-Encoded: Die Zustände des Controllers werden hierbei entsprechend ihrer Abfolge in der Protokollverarbeitung mit einem Binärcode numeriert. Vor dieser Zustandscodierung wird das Protokoll auf der Basis eines Zustandsgraphen analysiert und optimiert. Auf diese Weise wird eine minimale Anzahl an Zustandsregistern erreicht, deren Ansteuerung und Auswertung jedoch wegen des Binärcodes einen vergleichsweise hohen Aufwand an Logik und Laufzeiten erfordert.

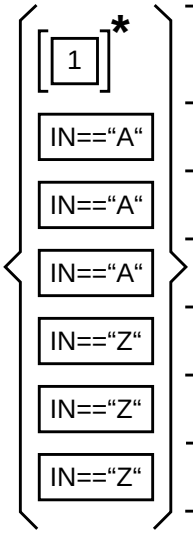
Min-Distance Min-Encoded: Wie bei Binary Min-Encoded werden die Zustände des Controllers zunächst über einen Zustandsgraphen bestimmt, ihre Codierung erfolgt aber mit minimierter Hammingdistanz der Zustandscodes in ihrer Abfolge, ähnlich dem Schema eines Gray-Codes. Hierdurch wird die Anzahl der Zustandsregister und die Anzahl der Bitwechsel pro Takt reduziert. Im Vergleich zu Binary Min-Encoded kann hierdurch für einige Zieltechnologien der Energieverbrauch der Schaltung und die Menge an Zustandslogik verringert werden.

Branch-Sequence Encoded: Der Zustandsgraph des Protokolls wird nach seiner Aufstellung und Optimierung hierbei in Teilgraphen partitioniert, die sich lokal gut für eine Binärcodierung eignen. Die Codierung eines Zustandes wird danach durch Konkatenation aus dem Binärcode seines Teilgraphen und dem Binärcode des Zustandes innerhalb des Teilgraphen gebildet. Dieser Zustandscode erfordert meist mehr Zustandsbits, als eine globale Binärcodierung, reduziert aber dafür die Menge an benötigter Zustandslogik und deren Laufzeiten.

One Hot Encoded: Die mittels eines Zustandsgraphen bestimmten Zustände des Protokolls werden hierbei auf einen 1-aus-n-Code abgebildet. Jeder Zustand wird hierbei durch ein Bit des Zustandsregisters repräsentiert, in dem nur jeweils ein Bit gesetzt ist (one hot). Diese Codierung benötigt eine vergleichsweise große Menge an Zustandsregistern, die jedoch keine redundanten Bits beinhaltet und deswegen oft kleiner als bei Distributed Encoded ist. Die Zustandslogik zeichnet sich auch hier durch Einfachheit und Schnelligkeit aus.

Beispiel 5.4 Die in Bild 5.8 vertikal dargestellte Frame-Definition sucht die Zeichenfolge „AAAZZZ“ auf dem Eingangsport IN. Die Anwendung verschiedener Controllercodierungen auf dieses einfache Protokoll ergibt die neben den Frames aufgeführten Zustandsvektoren. Es ist erkennbar, daß die meisten Zustandsbits

bei Distributed Encoded und One Hot Encoded benötigt werden, dafür aber beide Codierungen die Zustandsauswertung durch einfache Bitabgriffe erlauben. Bei den Codierungen Binary Min-Encoded und Min-Distance Min-Encoded werden die wenigsten Zustandsbits benötigt, von denen aber sowohl für die Auswertung als auch für die Fortschaltung der Zustände immer alle Bits gleichzeitig beachtet werden müssen. Für Branch-Sequence Encoded sind mehr Zustandsregister als bei den Min-Encoded Codes nötig. Dafür sind in der Zustandslogiken für die einzelnen Teilgraphen des Zustandsbaumes aber entsprechend weniger Bits zu beachten, so daß diese kleiner und schneller sind als die Min-Encoded Logiken.



Distributed Encoded	Binary Min-Encoded	Min-Distance Min-Encoded	Branch-Sequence Encoded	One Hot Encoded
0000001	000	000	00_00	0000001
0000011	001	001	01_00	0000010
0000111	010	011	01_01	0000100
0001111	011	010	01_10	0001000
0010001	100	110	10_00	0010000
0100001	101	111	10_01	0100000
1000001	110	101	10_10	1000000

Bild 5.8: Verschiedene Controller-Codierungen im Vergleich

5.3.2 Wiederholungen von Frames mit Zählern und deren Implementierung

Die Implementierung von Frame-Wiederholungen mit einer festgelegten Anzahl an Läufen erfolgt standardmäßig durch je einen automatisch angelegten Binärzähler für jeden Wiederholungsoperator. Dieser verwaltet die Laufindizes und steuert in Verbindung mit den ihn kontrollierenden Controllerzuständen die Frame-Wiederholungen.

Diese Implementierung führt bei mehreren gleichartigen Wiederholungen, die sich in der zeitlichen Ausführung nicht überschneiden können, zu einem unnötig hohen Aufwand an Zählerlogik. Alternativ können Wiederholungsoperatoren eine Zählervariable gemeinsam nutzen, die wie jede andere Protokollvariable vom Entwickler mit Bezeichnung und Bitbreite zu deklarieren ist (Bild 5.9).

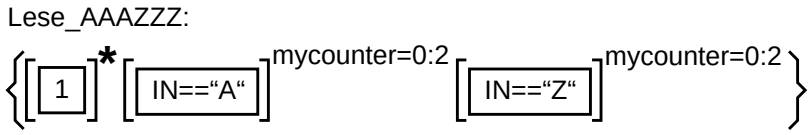


Bild 5.9: Frame-Wiederholungen mit einer Zählervariablen

Die Steuerung der gemeinsam genutzten Zählervariablen erfolgt standardmäßig ebenfalls als Binärzähler. Ein Binärzähler besitzt jedoch den Nachteil, aufgrund der für das inkrementelle Zählen benötigten Carry-Logik hohe Laufzeiten zu erfordern. Über ein Frame-Attribut ist alternativ ein *linear feedback shift register* (LFSR) wählbar, das in vielen Schaltungsentwürfen zur Generierung sogenannter Pseudozufallszahlen verwendet wird. Dieses durchläuft beginnend bei einem von Null verschiedenen Startwert alle Binärwerte seiner Bitbreite, ausgenommen den Nullvektor, in einer zufällig erscheinenden Reihenfolge. Da sich der LFSR-Wert nach einer bestimmten Anzahl von Takten ausgehend von einem vorgegebenen Startwert berechnen läßt, kann dieser LFSR-Wert als Markierung für das Ende des benötigten Zählvorgangs benutzt werden. Im Vergleich zu einem Binärzähler wird für die Zählung jedoch keine zeitaufwendige Carry-Logik benötigt, sondern nur eine XOR-Verknüpfung über zwei Eingänge (Bild 5.10).

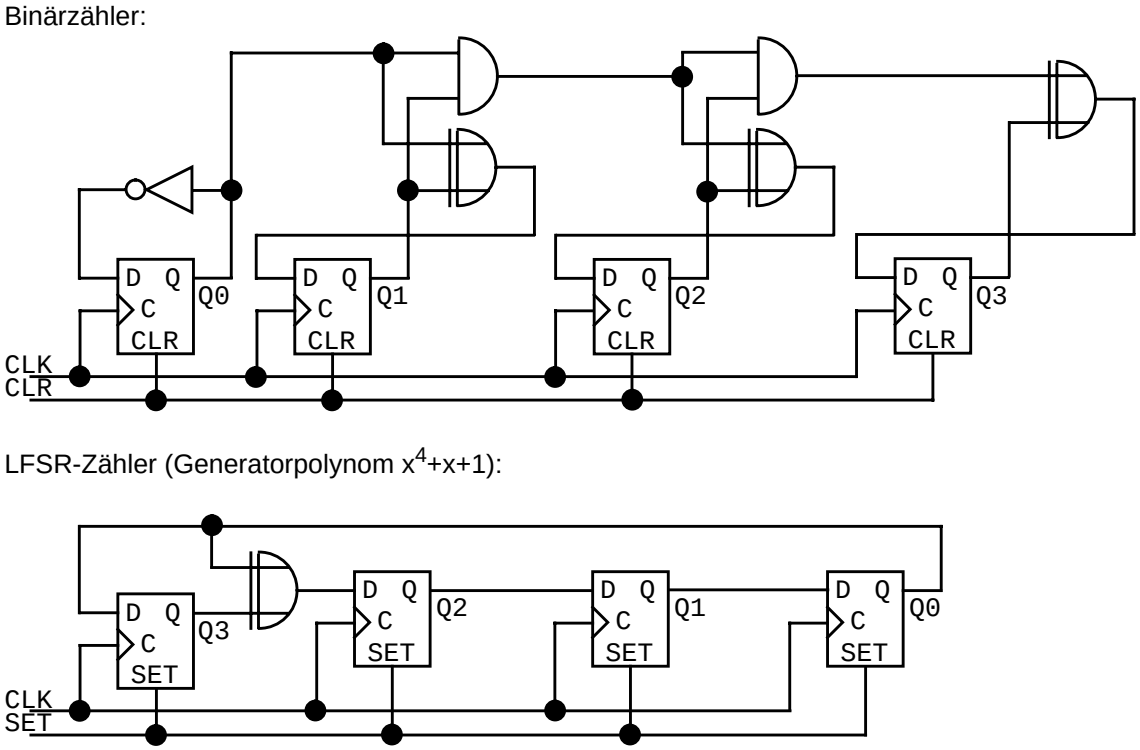


Bild 5.10: 4-Bit Binärzähler im Vergleich mit 4-Bit LFSR-Zähler

5.3.3 Implementierung von Berechnungen

Protokolldefinitionen können Berechnungen in den Ausführungsbedingungen von Frames, in Frame-Actions und Default-Actions direkt enthalten oder mittels HDL-Aufrufen einbinden. Die Implementierung der Berechnungen erfolgt grundsätzlich in Form kombinatorischer Logiken, deren Ausführung innerhalb eines Taktes des Controllers abzuschließen ist. Die Komplexität der Berechnungen und damit ihre kombinatorischen Laufzeiten sind neben der Zustandslogik der Controller-FSM bestimmend für die maximale Taktrate des Controllers, die jedoch erst im Rahmen einer RTL-Synthese abgeschätzt werden kann.

Eine Optimierung der Berechnungslaufzeiten auf der Ebene des Protokolls ist nur manuell durch den Entwickler möglich, indem dieser langsame Berechnungen in Teile zerlegt, die als Frame-Actions über sequentiell aufeinanderfolgende bzw. parallel ablaufende Frames verteilt werden. Die Bestimmung und Optimierung zeitkritischer Berechnungen ist aufgrund des rein manuellen Verfahrens und der in der Regel mehrfach zu durchlaufenden Arbeitsschritte von Protokollsynthese, Codeerzeugung und RTL-Synthese sehr arbeitsaufwendig.

Ist die Menge an Berechnungslogik ein kritisches Element im Entwurf, so ist zu beachten, daß jede Referenzierung einer Berechnung, direkt als Action oder indirekt über Referenzierung einer sie enthaltenden Frame-Definition, eine Kopie der Berechnungslogik erzeugt. Ist eine mehrfache Implementierung für parallele Ausführbarkeit aufgrund des Protokolls nicht erforderlich, so kann der Entwickler die Protokolldefinition diesbezüglich manuell optimieren. Anstelle der mehrfachen Referenzierung der Berechnung ist hierfür eine einzige Referenz in einer autonom arbeitenden Frame-Definition zu verwenden, die über Handshake-Signale und globale Variablen des Protokolls mit den Nutzer-Frames kommuniziert. Wie bei der Laufzeitoptimierung sind auch hier die Protokollsynthese, Codeerzeugung und RTL-Synthese meist mehrfach auszuführen und somit das Verfahren sehr arbeitsaufwendig.

5.4 Optimierungsmöglichkeiten bei der Controllersynthese

Insgesamt erlaubt die Controllersynthese mit dem Protocol-Compiler ein breites Spektrum an Optimierungen, welches von der vollautomatischen Reduzierung der Controllerzustände und deren Codierung bis zur rein manuellen Bestimmung von Zählervariablen und der Aufteilung von Berechnungen reicht [SynPCU99].

Zu den automatischen Optimierungsmöglichkeiten der Controllersynthese, die dem Entwickler weitgehend verborgen bleiben und lediglich durch Synthesemeldungen sichtbar werden, gehören:

- Aufstellung und Vereinfachung von Zustandsgraphen
- Zustandsanalyse zur Identifizierung redundanter und irrelevanter Frames

Durch Optionen der Dialogfenster bzw. Protokollattribute können folgende weitere Optimierungen für die Controllersynthese bestimmt werden:

- Aufteilung des Protokolls in getrennt optimierte und codierte Partitionen
- Auswahl der zu verwendenden Zustandscodierung
- Flächen- oder Laufzeitreduzierung der Controller-FSM
- Benutzung von LFSR-Zählern
- Unterdrückung von Pipelining in Ausführungspfaden
- Auswahl der Architektur für die Controller-FSM

Durch die Modellierung des Protokolls hat der Entwickler weitere Möglichkeiten, gezielt auf das Ergebnis der Protokollsynthese Einfluß zu nehmen:

- Mehrfache Benutzung eines Zählers für Frame-Wiederholungen

- Laufzeitreduzierung durch Aufteilung von Berechnungen
- Flächenreduzierung durch Referenz-Reduzierung
- Unterdrückung von Pipelining durch entsprechende Frame-Bedingungen

Innerhalb der eingebundenen HDL-Funktionen und -Instanzen sind zudem alle in Kapitel 3 und [SynDCR97] genannten Modellierungsoptimierungen anwendbar.

6 Vergleich und Auswahl von Synthesemethoden

In den vorangegangenen Kapiteln wurden die Synthesemethoden RTL-, High-Level- und Controllersynthese in ihren grundlegenden Eigenschaften vorgestellt, welche aus den Spezifikationen der Eingabesprachen und Werkzeuge entnommen oder abgeleitet werden konnten. In diesem Kapitel werden auf der Grundlage dieser Eigenschaften zunächst Auswahlkriterien für geeignete Synthesemethoden zu Entwurfsaufgaben entwickelt und Möglichkeiten zu ihrer Anwendung erörtert.

Anhand der Ergebnisse mehrerer Entwurfsprojekte erfolgt eine genauere Ausarbeitung dieser Auswahlkriterien für die bei Abwägungen von Eigenschaften anzuwendenden Gewichtungen. Solche Auswertungen stellen einen wesentlichen Teil der im Rahmen dieser Arbeit vorgenommenen Forschungen dar.

Weiterhin werden die experimentellen Untersuchungen zum Entwurfsraum über Syntheseoptimierungen betrachtet. Ihre Anwendbarkeit und Auswirkungen im Vergleich zur Modellierung stehen dabei im Vordergrund.

6.1 Grundlegende Eigenschaften im Überblick

Die vorangegangenen Betrachtungen der RTL-Synthese (Kapitel 3), High-Level-Synthese (Kapitel 4) und Controllersynthese (Kapitel 5) behandelten die jeweiligen Eigenschaften dieser Synthesemethoden im Zusammenhang mit der Modellierung und der Einbettung in den Entwurfsablauf. Diese Eigenschaften werden hier aus der Perspektive der Entwurfsmöglichkeiten betrachtet und verglichen.

Die Entwurfsmöglichkeiten einer Synthesemethode werden sowohl durch die als Eingangsbeschreibung verwendbaren Modelle als auch durch ihre Umsetzung in Schaltungen bestimmt. Eine detaillierte Aufschlüsselung dieser Faktoren führt für die Eingangsbeschreibung zu den in Kapitel 2 genannten Eigenschaften einer Entwurfsbeschreibung:

- Zeitdarstellung (Laufzeiten, Takte und Protokolle)
- Datenrepräsentation (Signale, Bits und Zahlen)
- Datenverarbeitung (physikalisch, logisch und algorithmisch)
- Entwurfsebene (Logik-, RT-, Algorithmus und Systemebene)
- Strukturauflösung
  (Gatter, Register und kombinatorische Logik sowie Funktionsblöcke)
- Beschreibungsart (Logikterme, Algorithmen und Ablaufdiagramme)

Die Umsetzung in eine Schaltung besitzt folgende Teilaspekte:

- Festlegung einzelner Schaltungskomponenten für eine Zieltechnologie
- Berücksichtigung von speziellen Eigenschaften der Eingangsbeschreibung in einer Zieltechnologie (z.B. Reset, Logikstruktur, und -granularität)
- Anordnung von Komponenten in Hierarchien und Gruppen
- Optimierung der Schaltung in Struktur, Stromverbrauch und Testbarkeit
- Erweiterung der Schaltung um Testlogiken für den Produktionsendtest

Eine tabellarische Gegenüberstellung dieser Eigenschaften für die drei Synthesemethoden zeigt Tabelle 6.1.

Eigenschaft	RTL-Synthese	Protokollsynthese	High-Level-Synthese
Zeitdarstellung	Laufzeiten und Takte	Takte eines globalen Taktnetzes	Takte globaler Taktnetze
Datenrepräsentation	uni-/bidirektionale Signale, Bits sowie Vektoren und Felder davon	unidirektionale Signale, Bits sowie Vektoren davon	unidirektionale Signale, Bits und Zahlen sowie Vektoren und Felder davon
Datenverarbeitung	logisch, arithmetisch und algorithmisch	logisch und arithmetisch	logisch, arithmetisch und algorithmisch
Entwurfsebene	Logik-, RT-, und Algorithmusebene	RT-Ebene, da Frames bzw. Zustände taktgebunden und Aktionen kombinatorisch sind	RT-, und Algorithmusebene
Strukturauflösung	Gatter, Register und kombinatorische Logik sowie Funktionsblöcke	Register (für Variablen), Controller-FSM und eingebundener RTL-HDL-Code	FSMD und eingebetteter RTL-HDL-Code
Beschreibungsart	Boolesche und arithmetische Operationen sowie algorithmische Anweisungsfolgen	Produktionen, Graphen, Boolesche und arithmetische Operationen sowie RTL-HDL-Code	algorithmische Anweisungsfolgen und RTL-HDL-Code
Festlegung von Komponenten für eine Technologie	Exakt durch Instanzen, grob mit Modellierung kombinatorischer und sequentieller Logik	Nur über Instanzen für die anschließende RTL-Synthese	Nur über Instanzen für die anschließende RTL-Synthese
Berücksichtigung von besonderen Eigenschaften der Beschreibung in Zieltechnologie	Multiplexer und Arithmetik für Binärworte, Ausblendung von nicht synthetisierbarem Code für spezielle Hardware-Eigenschaft (z.B. Power-On-Reset)	keine	Abbildung von Array-Variablen auf RAMs
Anordnung von Komponenten	Exakt durch Instanzen, Anweisungsfolgen und Klammerung innerhalb von Anweisungen	Mit Anweisungsfolgen in eingebundenem RTL-HDL-Code	Mit Instanzen und Anweisungsfolgen in eingebettetem RTL-HDL-Code
Optimierung der Schaltung	kombinatorische Logik auf Fläche, Laufzeiten und Stromverbrauch; Gesamtschaltung auf Testbarkeit	FSM-Registeranzahl oder Komplexität der FSM-Zustandslogik mit Zustandscodierung bzw. Zählervarianten	Mehrfachnutzung von Registern und arithmetischen Operatoren sowie Taktanzahl und -periode
Testlogiken	Scanpfad	nicht möglich	nicht möglich

Tabelle 6.1: Gegenüberstellung von Eigenschaften der Synthesemethoden

Die Entwurfsmöglichkeiten einer Synthesemethode ergeben sich sowohl direkt aus diesen Eigenschaften als auch indirekt aus ihrem Zusammenspiel. So ermöglicht nur die RTL-Synthese bidirektionale Kommunikation über Signale sowie mittels ungetakteter, kombinatorischer Beschreibungen asynchrone Verarbeitung. Mit den anderen beiden Synthesemethoden ist dies wegen ausschließlich getakteter Zeitdarstellung bzw. unidirektionaler Signale nicht möglich.

Eigenschaften oder ihre Kombinationen, die nur für eine Synthesemethode vorhanden sind, zeichnen diese gegenüber den anderen Methoden aus und sind damit charakteristisch für sie. Im Falle der RTL-Synthese sind dies:

- Modellierung kombinatorischer Logik
- bidirektionale Kommunikationssignale
- Vorgabe der Schaltungsstruktur auf RT- und Gatterebene
- Register mit Pegelsteuerung oder beliebigen asynchronen Kontrolleingängen
- Exakte Festlegung von Komponenten der Zieltechnologie
- Optimierung des Stromverbrauchs
- Einfügen von Testlogiken

Die Controllersynthese zeichnet sich aus durch:

- weitgehend graphische Entwurfseingabe
- hierarchische Zustandsverarbeitung in Produktionen oder Graphen
- Exploration von FSM-Konstruktionen, -Codierungen und -Optimierungen

Charakteristisch für die High-Level-Synthese sind:

- Einteilung einer Anweisungsfolge in Takte und Zuordnung der Operationen
- Verwaltung von Registern und Operatoren als allgemeine Arbeitsressourcen

Erfordert eine Entwurfsaufgabe die sich aus einer charakteristischen Eigenschaft ergebenden Entwurfsmöglichkeiten zu ihrer Lösung, so ist die zu verwendende Synthesemethode bereits durch die Aufgabe festgelegt. Ist eine charakteristische Eigenschaft jedoch nicht zwingend gefordert, so ist die nach Prioritäten abgestufte Abwägung benötigter und angebotener Eigenschaften der Synthesemethoden zur Auswahl nötig.

Hierbei besitzen unverzichtbare Eigenschaften den höchsten Stellenwert, da ihr Fehlen ein sicheres Ausschlußkriterium für eine Synthesemethode ist. Sind Eigenschaften alternativ gegeneinander abzuwägen oder optional, so besteht eine starke Abhängigkeit der Auswahlkriterien von den Zielvorstellungen über den Entwurf. Ausschlaggebend sind hierbei aber nicht nur die Entwurfsergebnisse für:

- Schaltungsgröße
- Arbeitsgeschwindigkeit
- Leistungsverbrauch
- Testbarkeit

Entscheidend sind ebenso die Bedingungen und Vorgaben des Entwurfsumfeldes:

- maximale Entwurfsdauer
- vorhandene Werkzeuge
- Kosten neuer Werkzeuge (Software, Hardware und Anwendungstraining)
- wiederverwendbare Entwurfskomponenten
- Erfahrungen im Entwurfsablauf
- Nutzung der vorgesehenen Zieltechnologie

Die Bedingungen und Vorgaben des Entwurfsumfeldes haben in der industriellen, geschäftsorientierten Entwurfspraxis durch ihren Einfluß auf die Entwurfskosten in der Regel einen höheren Stellenwert, als bestmögliche Entwurfsergebnisse. Für die Ergebnisse ist hierbei lediglich wichtig, daß die Mindestanforderungen an den Entwurf erfüllt werden.

Für vergleichende Untersuchungen, deren Ergebnisse eine Hilfestellung für zukünftige Entwurfsentscheidungen liefern sollen, stellt ein solcher Einfluß des Entwurfsumfeldes ein Problem dar. Um Ergebnisse und die darauf aufbauenden Folgerungen auf andere Entwürfe anwenden zu können, wäre ein standardisiertes Entwurfsumfeld nötig. Dies ist aber sowohl wegen des für jeden Entwurf und seine Entwickler unterschiedlichen Umfeldes als auch wegen der Weiterentwicklung von Entwurfserfahrung der Entwickler, von Werkzeugen und von Zieltechnologien praktisch nicht realisierbar.

In dieser Arbeit wird der Einfluß des Entwurfsumfeldes daher weitgehend vernachlässigt. Neben den Ergebnissen für Schaltungsgröße und Arbeitsgeschwindigkeit, die für alle Entwürfe genau bestimmbar und untereinander vergleichbar sind, wird nur die grobe Auswirkung auf die Entwurfsdauer aufgrund ihrer großen Bedeutung in der Entwurfspraxis berücksichtigt. Für die Auswahl einer Synthesemethode zu einer Entwurfsaufgabe ergeben sich unter dieser Voraussetzung als übertragbare Auswahlkriterien mit einer der Auflistungsabfolge entsprechenden Bedeutung:

1. Entwurfseigenschaften, die charakteristisch für eine Synthesemethode sind
2. Eigenschaften einer Synthesemethode und ihre Kombinationen, welche die Anforderungen an Schaltungsgröße und -geschwindigkeit mit möglichst kurzer Entwurfsdauer erfüllen

Im ersten Fall ist die Auswahl durch die benötigten Entwurfsmöglichkeiten klar festgelegt und beschränkt sich auf die Prüfung von Eigenschaften, welche durch die Aufgabenstellung gefordert sind. Für den zweiten Fall ist die Beschreibung der Auswahl über erreichte Ergebnisse, die erst nach einer Entwurfsdurchführung bekannt sind, aber keine praktisch anwendbare Hilfe. Für eine korrekte Auswahl einer Synthesemethode vor einer Entwurfsdurchführung wäre hierbei eine auf die jeweilige Aufgabenstellung abgestimmte Gewichtung der einzelnen Eigenschaften einer Synthesemethode als Entscheidungsgrundlage erforderlich.

Dies setzt eine Klassifizierung von Entwürfen anhand ihrer Aufgaben und der mit unterschiedlichen Synthesemethoden erzielbaren Ergebnisse voraus, unter der sich aus dem Vergleich der Ergebnisse ergebende Gewichtungen einordnen

lassen. Die hierfür benötigten Ergebnisse wurden in mehreren Entwurfsprojekten mit zahlreicher Synthesevariationen bestimmt und werden zusammen mit den aus ihnen gezogenen Folgerungen im nächsten Abschnitt vorgestellt.

6.2 Empirische Betrachtung der Synthesemethoden

Dem im vorigen Abschnitt erkennbar gewordenen Mangel an aufgabenabhängigen Gewichtungen von Entwurfseigenschaften zur Auswahl einer Synthesemethode wird in diesem Abschnitt mit einer empirischen Betrachtung begegnet.

Hierzu werden mehrere Entwurfsaufgaben, die an der Abteilung E.I.S. für Vorlesungsbeispiele, Praktika, Studien- und Diplomarbeiten sowie für dieses Forschungsprojekt entwickelt wurden, mit den verschiedenen Synthesemethoden in zahlreichen Synthesevarianten untersucht. Die Entwurfsanforderungen und die mit den Methoden erzielten Ergebnisse bilden die Basis für die Klassifizierung der Entwürfe im nachfolgenden Abschnitt, unter welcher die Gewichtungen für die vereinfachte Auswahl einer Synthesemethode eingeordnet werden können. Die Entwurfsaufgaben werden zunächst in eigenen Unterabschnitten betrachtet, wobei neben den Entwurfsanforderungen die untersuchten Synthesemethoden und die mit ihnen erzielten Ergebnisse beschrieben werden.

Auf bekanntere Referenz-Entwürfe, wie die RTL- und High-Level-Synthese-Benchmarks der High-Level Synthese Workshops (HLWS) bzw. Logic Synthesis Workshops (IWLS) nach [HLSynt89], [HLSynt91], [HLSynt92], [HLSynt95], [LGSynt89], [LGSynt91] und [LGSynt93] wurde nicht zurückgegriffen. Diese entsprechen einerseits in ihrer Beschreibung zum Teil nicht den Formen für die RTL- bzw. High-Level-Synthese und sind andererseits wenig dokumentiert, so daß die Verifikation der korrekten Umsetzung und die Beurteilung der Qualität der Beschreibungen kaum möglich sind. Zudem gibt es zu diesen Benchmarks, anders als bei den untersuchten Entwürfen, keine alternativen Implementierungen für den Vergleich von Ergebnissen innerhalb und zwischen den Synthesemethoden.

Die Untersuchungen erfolgten zur Gewinnung einer aussagekräftigen Datenbasis mit automatisiert parametrisierten und gesteuerten Abläufen für Synthese, Plazierung und Verdrahtung. Nur so waren die bis zu 2300 Variationsmöglichkeiten pro Synthesemethode durchführbar. Eine zentrale Rolle spielte hierbei die Skriptsprache *perl* [WalSch91], welche die Erstellung der Parameterdateien, die Steuerung der Synthese- und FPGA-Werkzeuge sowie die Auswertung der jeweils erzielten Ergebnisse in der benutzten UNIX-Umgebung problemlos ermöglichte.

6.2.1 Entwurf eines Bildschirmcontrollers

Der hier untersuchte Bildschirmcontroller wurde aus dem Vorlesungsbeispiel discount entwickelt, welches den FPGA-Entwurf vollständig, aber ohne Synthese behandelt [Golze99]. Neben einer RTL-Synthese des modifizierten discount-Modells wurden hierfür in einer Diplomarbeit die RTL-Synthese und die High-Level-Synthese durchgeführt [Friedr98]. Die Ergebnisse werden durch die Controllersynthese vervollständigt. Die an den synthetisierten Entwurf gestellten Anforderungen umfaßten für jede betrachtete Synthesemethode:

- Taktung mit Bildpunktrate von 14.32MHz
- Im Taktraster festgelegtes I/O-Protokoll für den Videodaten-Bitstrom
- Implementierung auf möglichst kleinem und langsamen Xilinx-3000-FPGA
- Fest vorgegebenes, zyklisches Verarbeitungsschema

Da sich zwischen den Ergebnissen der Synthesemethoden deutliche Unterschiede ergaben, konnte die vorgegebene Taktung bei möglichst kleinem und langsamen FPGA nicht für alle Schaltungen mit demselben FPGA-Typ erfolgen. Um einen direkten Vergleich zwischen den Ergebnissen dennoch zu ermöglichen, wurden die Syntheseläufe zusätzlich für ein einheitliches FPGA (3090A-7) durchgeführt.

RTL-Synthese des Bildschirmcontrollers: Die für die RTL-Synthese verwendeten Modelle des Bildschirmcontrollers entstanden durch kleinere Modifikationen des Vorlesungsbeispiels [Golze99] bzw. durch eine Neukonstruktion auf der Basis der Spezifikation [Friedr98]. Die Modelle lösen die Struktur feinkörnig in Register und kombinatorische Logik auf. Die Entwurfsdauer betrug jeweils etwa einen Tag.

Zusätzlich zu [Friedr98] wurden neben den besten Ergebnissen Variationen durch verschiedene Optimierungen der Synthesewerkzeuge untersucht. Variiert wurden die Einstellungen für:

- Taktvorgabe
- Flächenbegrenzung (`set_max_area`)
- Strukturoptimierungen (`set_structure`, `boolean`, `timing`)
- Übersetzungsoptionen (`incremental_map` und `prioritize_min_paths`)

Durch die RTL-Synthese und die daran anschließende Abbildung auf ein 3042-125 FPGA entstand ein breit gestreuter Ergebnisbereich für Logikverbrauch (`CLBs`) und erreichte Taktrate (`MHz`), der in Bild 6.1 mit einer Kennzeichnung der Modelle dargestellt ist. Es ist deutlich erkennbar, daß sich die Ergebnisbereiche nicht überschneiden und das Modell nach [Golze99] einen größeren Ergebnisspielraum bei meist höherer Taktrate und größerem Logikverbrauch besitzt. Die Ergebnisse und die verwendeten Syntheseeinstellungen sind in Tabelle A.2 des Anhangs A aufgelistet.

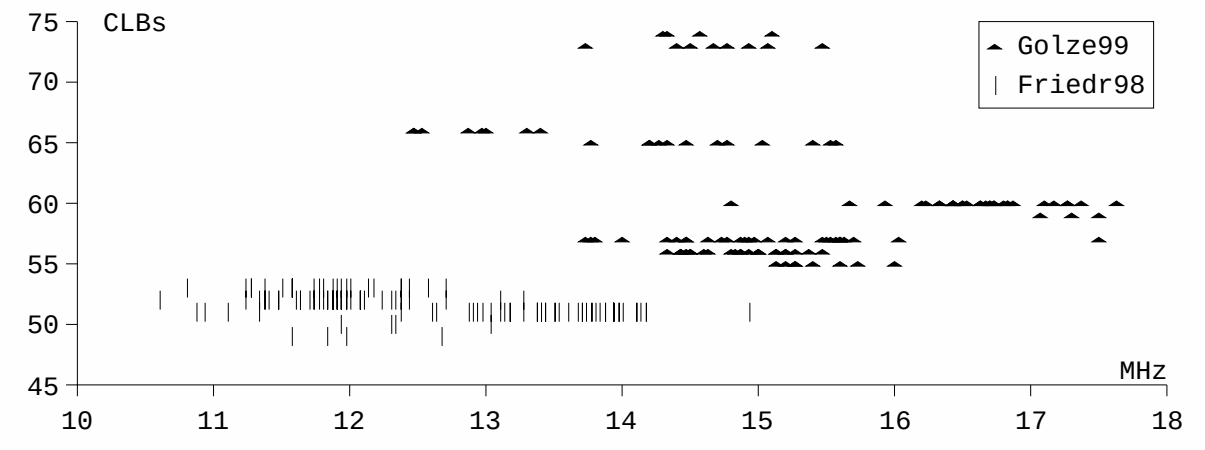


Bild 6.1: RTL-Modellierungsvarianten des discount

Die beiden Modelle sind sich in Modularisierung und verwendeten Operationen, im wesentlichen Inkrementer und Vergleicher, sehr ähnlich. Als Ursache für die unterschiedlichen Ergebnisspielräume ist daher das abweichende Verhältnis zwischen kombinatorischer und sequentieller Logik anzunehmen. Der Entwurf nach [Golze99] verwendet mehr Register zur Aufteilung von Berechnungspfaden als [Friedr98]. Als Konsequenz davon müssen sich kürzere kombinatorische Laufzeiten und damit höhere Taktraten ergeben. Gleichzeitig stehen die Register einer optimalen Füllung der CLB-Lookup-Tables im Wege, so daß mehr CLBs für die Implementierung der Logik notwendig sind.

In Bild 6.1 wird der im Vergleich zu den Syntheseeinstellungen dominierende Einfluß der Modellierung für diesen Entwurf deutlich. Der Einfluß der einzelnen Syntheseeinstellungen in einem Ergebnisbereich wird bei Kennzeichnung der aus bestimmten Einstellungen folgenden Ergebnisse erkennbar. In Bild 6.2 wird nach Flächenbegrenzung unterschieden, in Bild 6.3 nach Strukturoptimierungen, in Bild 6.4 nach `compile`-Optionen und in Bild 6.5 nach der Taktvorgabe.

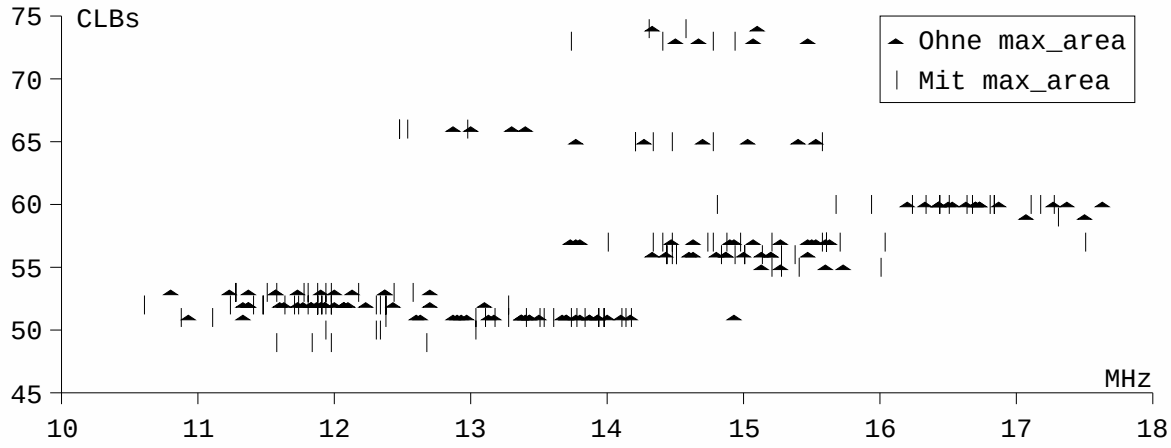


Bild 6.2: `set_max_area` bei der RTL-Synthese des discount

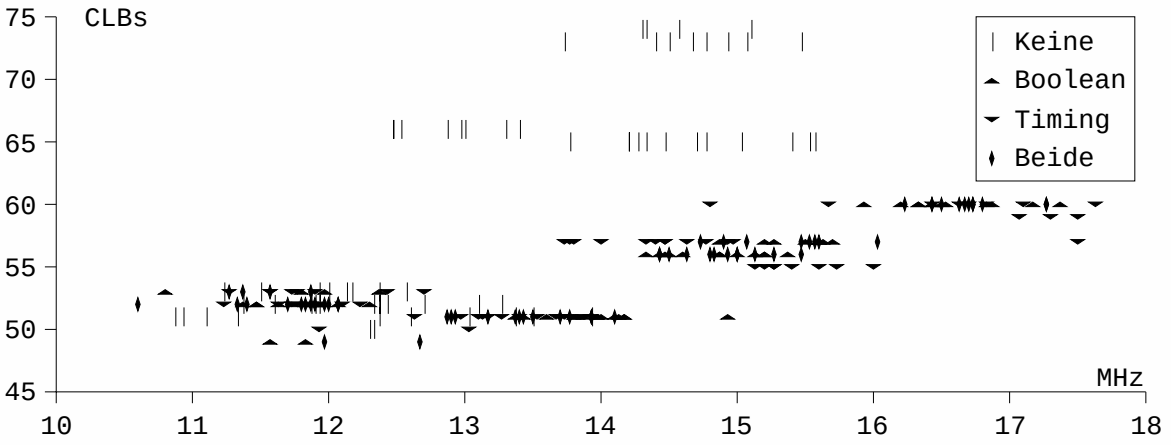


Bild 6.3: Strukturoptimierungen bei der RTL-Synthese des discount

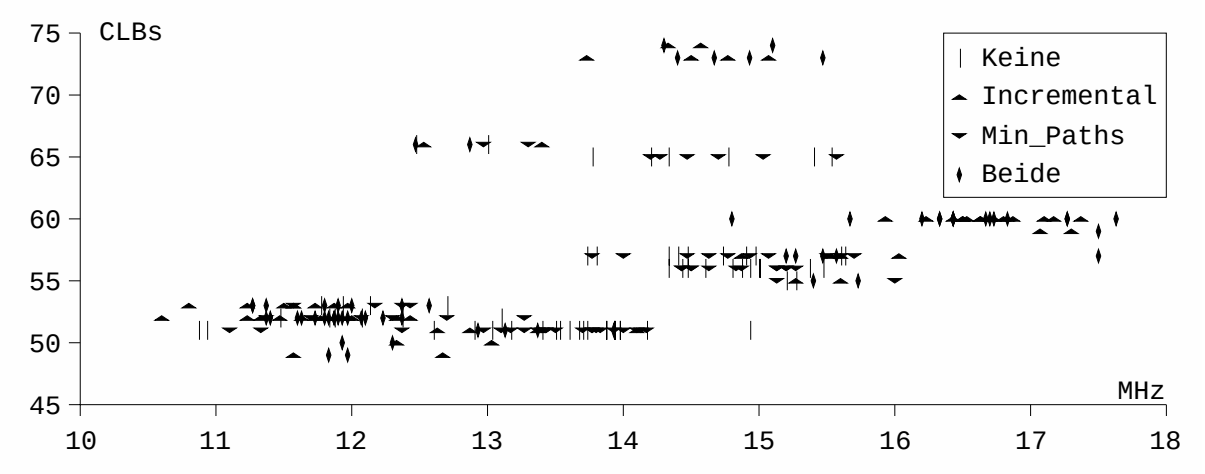


Bild 6.4: `compile`-Optionen bei der RTL-Synthese des discount

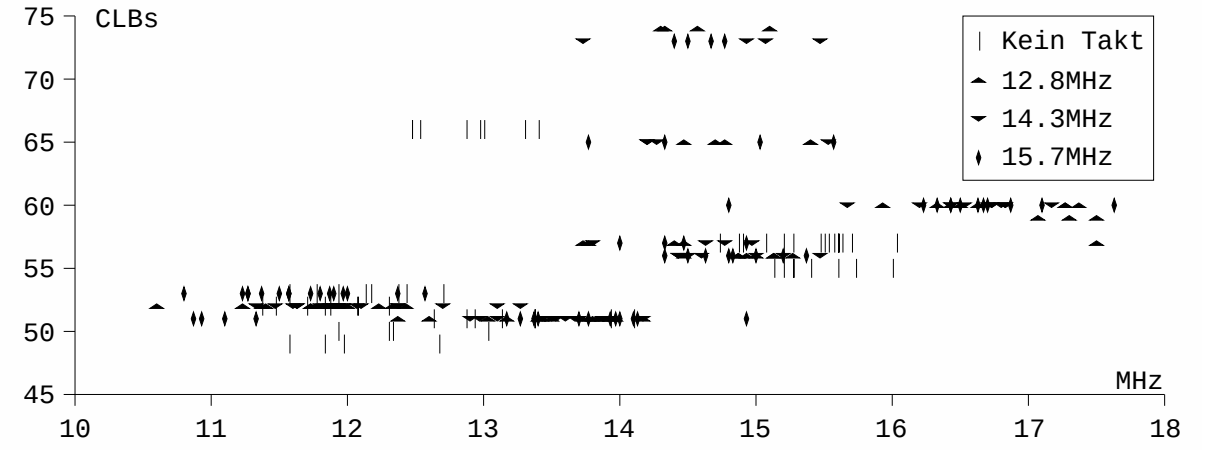


Bild 6.5: Taktvorgaben bei der RTL-Synthese des discount

Während sich aus der Verteilung der Ergebnisse für die Flächenbegrenzung und die `compile`-Optionen keine für beide Entwürfe zutreffende Tendenz erkennen läßt, bestehen für die Strukturoptimierungen und die Taktvorgaben sichtbare Zusammenhänge zwischen Syntheseeinstellungen und Ergebnisbereichen. So sind hohe Taktraten nur im Falle einer Taktspezifikation vorhanden, wobei die um 10% vom Sollwert abweichenden Spezifikationen geringfügig schnellere und kleinere Ergebnisse liefern als eine exakte Spezifikation. Die kleinsten Ergebnisse werden ohne Taktvorgaben erreicht. Bei den Strukturoptimierungen fällt auf, daß sowohl die kleinsten als auch die schnellsten Ergebnisse durch einzelne oder kombinierte Anwendung von Boolescher und Timing-Optimierung erreicht werden, während ohne diese Optimierungen viele Ergebnisse langsamer und größer sind als das jeweilige Spitzenfeld.

High-Level-Synthese des Bildschirmcontrollers: Das Modell für die High-Level-Synthese des Bildschirmcontrollers stammt aus [Friedr98] und ist durch die Verwendung von Zählvariablen und die Vermeidung von Schleifen einem RTL-

Modell sehr ähnlich. Diese Modellierungsart ermöglichte unter den getesteten Alternativen als einzige die Erfüllung des taktgenau vorgegebenen I/O-Protokolls. Durch die Festlegung auf ein `cycle_fixed`-Scheduling mit einem Takt für jeden Prozeßdurchlauf bestanden keine Variationsmöglichkeiten für Scheduling und Allocation der High-Level-Synthese. Bedingt durch fehlerhafte Entwurfsansätze, die zu keinem erfolgreichen Scheduling führten, wurde für die High-Level-Beschreibung eine Entwurfszeit von etwa 3 Tagen benötigt.

Bei der abschließenden RTL-Synthese des Entwurfsablaufs konnten bis auf die Taktvorgabe, die für ein erfolgreiches Scheduling feststand, verschiedene Einstellungen wie beim direkten RTL-Entwurf verwendet werden. Die Ergebnisse dieser im folgenden graphisch ausgewerteten Syntheseläufe für den FPGA-Typ 3164A-2 sind in Tabelle A.3 zu finden.

Die Kennzeichnung der `compile`-Optionen in Bild 6.6 läßt eine Aufteilung des Ergebnisbereiches durch `incremental_map` deutlich werden, wobei mit dieser Option signifikant langsamere und gleichzeitig größere Schaltungen entstehen als ohne sie. Als Ursache ist eine Unverträglichkeit zwischen High-Level-Synthese und `incremental_map` bei diesem Entwurf zu vermuten, weswegen die mit der Option erzielten, sehr schlechten Ergebnisse nicht genauer betrachtet werden.

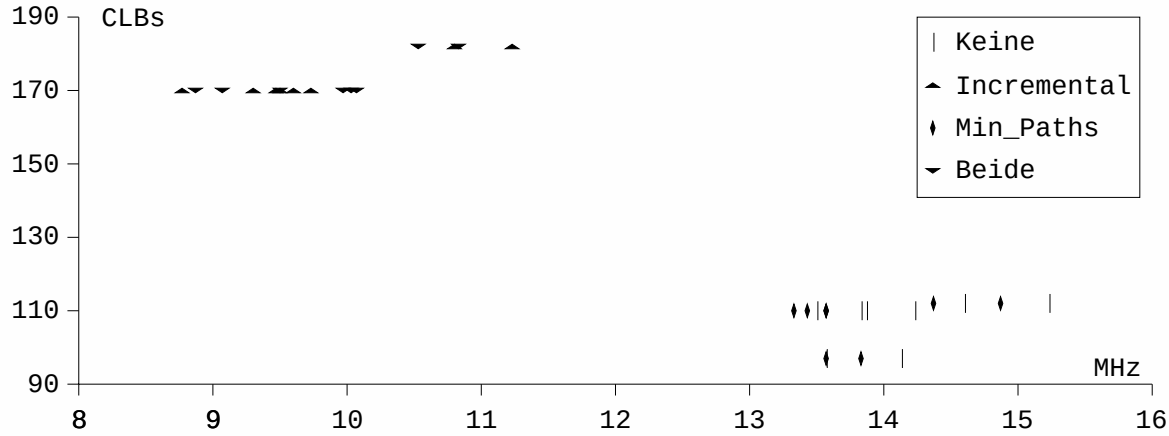


Bild 6.6: `compile`-Optionen nach der High-Level-Synthese des discount

Die Option `prioritize_min_paths`, die im folgenden mit `min_paths` bezeichnet wird, hat keine deutlich erkennbaren Einflüsse. Eine genauere Darstellung der Ergebnisse ohne `incremental_map` bei Kennzeichnung der Flächenbegrenzung (Bild 6.7) läßt auch für `set_max_area` keine Beziehung zur Schaltungsgröße oder -geschwindigkeit sichtbar werden. Die Ergebnisbereiche überlagern sich ohne eindeutige Tendenz und die Unterschiede der jeweiligen Randwerte liegen im Rahmen der Plazierungs- und Verdrahtungsstreuung.

Die Verwendung von Strukturoptimierungen (Bild 6.8) führt generell zu kleinen und langsamen Schaltungen, wobei eine reine Timingoptimierung die kleinsten Ergebnisse liefert. Ohne diese Optimierungen werden die schnellsten aber auch größten Schaltungen erzielt. Eine Aktivierung der Booleschen und der Timing-Optimierung bewirkt ebenso wie die rein Boolesche Strukturierung etwas

kleinere Ergebnisse als keine Optimierung bei gleichzeitigen Einbußen in der verwendbaren Taktrate.

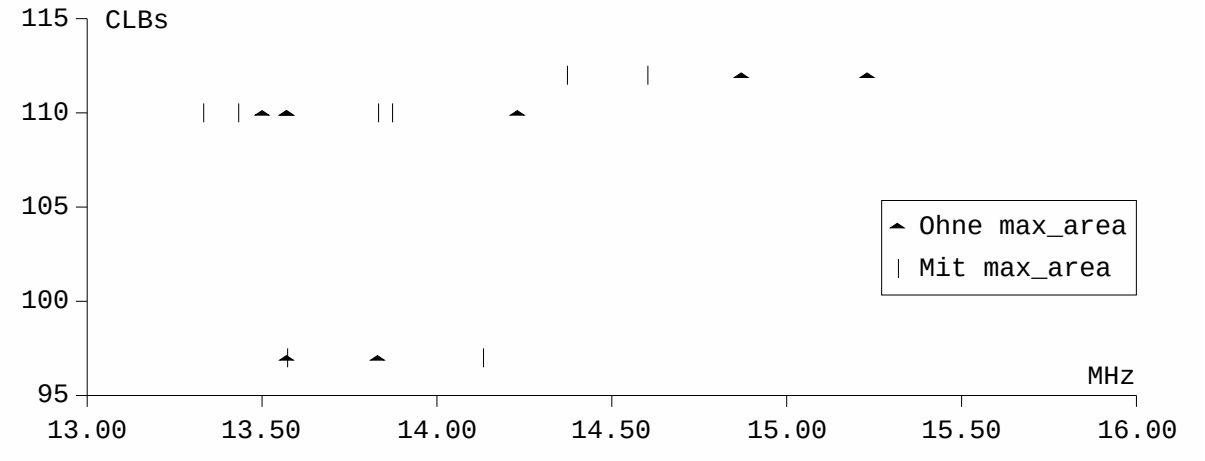


Bild 6.7: `set_max_area` nach der High-Level-Synthese des discount

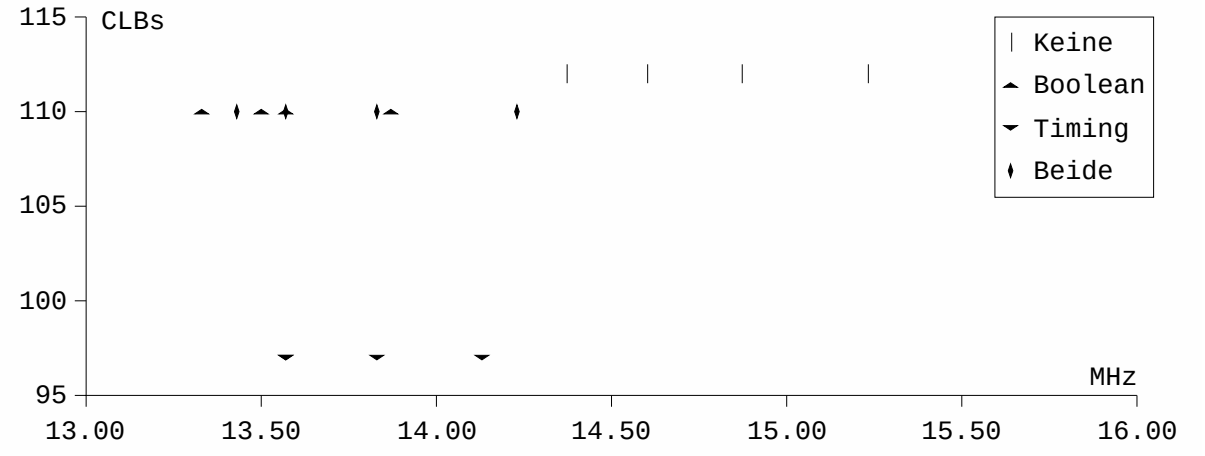


Bild 6.8: Strukturoptimierungen nach der High-Level-Synthese des discount

Verglichen mit den Auswirkungen der Einstellungen für die direkte RTL-Synthese sind ähnlichen Tendenzen der Ergebnisse bei gleichen Synthesevariationen nur für die Strukturoptimierungen festzustellen. Diese verringern in beiden Fällen die resultierende Schaltungsgröße, beeinflussen die Schaltungsgeschwindigkeit aber unterschiedlich.

Controllersynthese des Bildschirmcontrollers: Die Modellierung des Bildschirmcontrollers mit der Protocol-Compiler-HDL besteht aus einer Endloswiederholung des Bildaufbauprotokolls, das sich aus 314 Wiederholungen des Zeilenprotokolls zusammensetzt (Bild 6.9). Die Verwaltung des Bildbereiches wird durch das Zeilenprotokoll und durch eine Zählvariable für die Zeilennummer vorgenommen. Die kompakte Modellierung des gesamten Bildaufbaus in einem Frame führte zu einer Entwurfszeit von etwa einen halben Tag. Die Controllersynthese bietet auf der Grundlage dieses Modells viele Möglichkeiten zur Controllervariation durch:

- Zustandscodierungen

- HDL-Modellstrukturen (Single-, Split oder Multi-Process-Architecture)
- LFSR-Zähler oder gemeinsamer Binärzähler für die vier Zeilenabschnitte

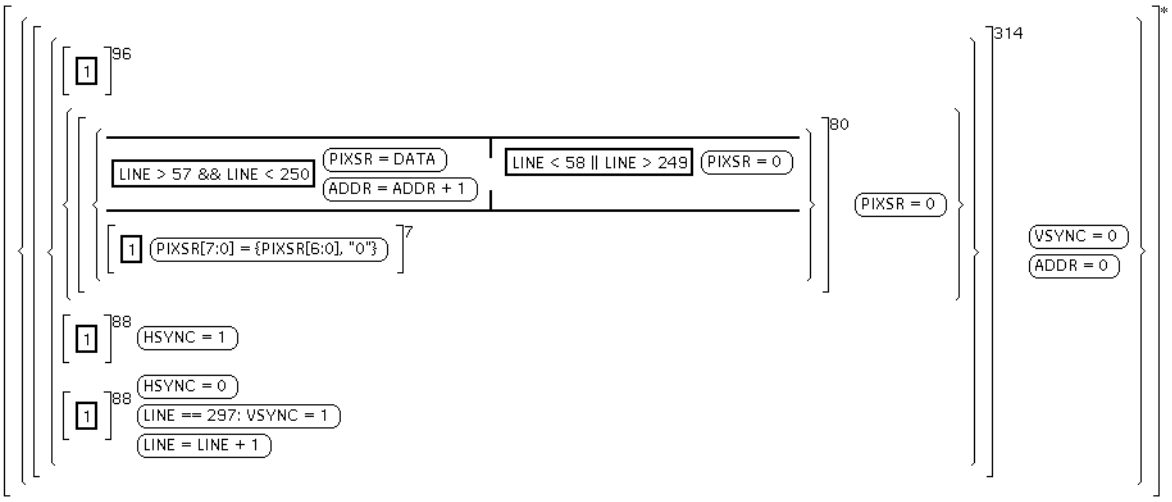


Bild 6.9: Protocol-Compiler-Modell des Bildschirmcontrollers

Controllervarianten ohne Zähler für Wiederholungen wurden wegen der hierbei undurchführbaren Zustandsanalyse nicht untersucht. Die Controllervarianten wurden im Anschluß mit der RTL-Synthese behandelt. Eine Kennzeichnung der in Tabelle A.4 zusammengefaßten Ergebnisse auf dem FPGA-Typ 3090A-7 nach Zustandscodierungen zeigt den Einfluß der Codierungsvarianten (Bild 6.10).

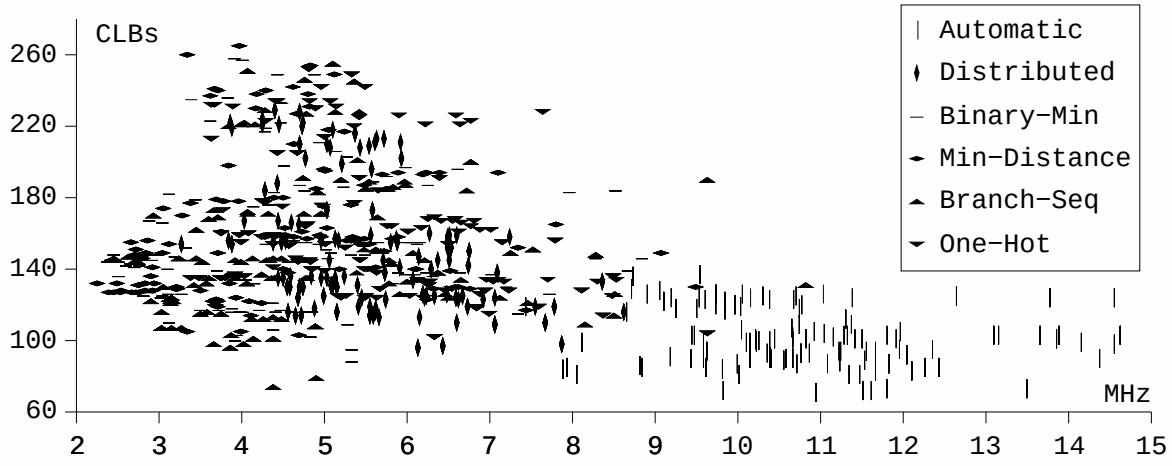


Bild 6.10: Codierungsvarianten der Controllersynthese des discount

Die automatische Partitionierung und Zustandscodierung des Controllers hebt sich durch geringere Größe und höhere Geschwindigkeit deutlich von den übrigen Ergebnissen ab. Hierbei wurde die 80 mal wiederholte Schleife für den Bildbereich einer Zeile als Min-Distance-Partition implementiert, während die umgebenden Konstrukte in eine Distributed-Partition umgesetzt wurden. Die verbleibenden Codierungsvarianten besitzen keine derart gerichtete Wirkung auf die Ergebnisse.

Eine Unterscheidung der Ergebnispunkte nach Prozeßarchitekturen der generierten Verilog-Modelle (Bild 6.11) zeigt für die Single Process Architecture eine Tendenz zu kleineren Schaltungen als bei den anderen beiden Strukturen.

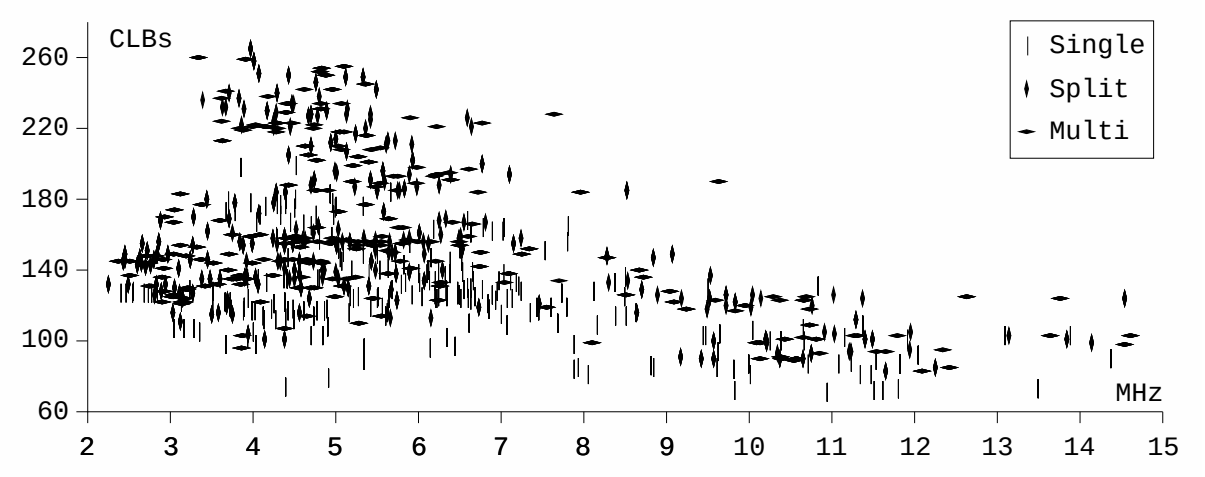


Bild 6.11: HDL-Modellstrukturen der Controllersynthese des discount

Auf die Geschwindigkeit ist kein Einfluß der Prozeßarchitektur feststellbar. Bei der Kennzeichnung der Zählertypen (Bild 6.12) werden die schnellsten Ergebnisse einer bestimmten Größe immer mit LFSR-Zählern erreicht, während Binärzähler für eine Geschwindigkeitsstufe die kleinsten Lösungen liefern. Die Wahl eines Zählertyps ist wegen der großflächigen Streuung und Überlappung der Ergebnisse aber nur als Voraussetzung für schnelle bzw. kleine Schaltungen zu sehen, jedoch nicht als Garant für sie.

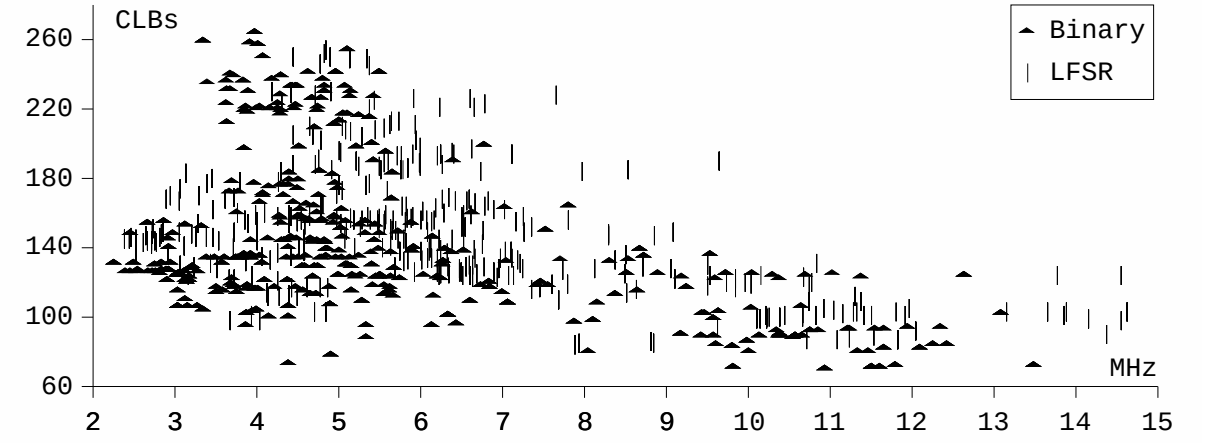


Bild 6.12: Zählervarianten der Controllersynthese des discount

Die im Anschluß an die Controllersynthese mit dem Protocol-Compiler folgende RTL-Synthese läßt für die in Bild 6.13 hervorgehobene Flächenvorgabe wie schon bei der direkten RTL-Synthese und der High-Level-Synthese keine Beziehung zur Schaltungsgröße oder -geschwindigkeit sichtbar werden.

Bei den in Bild 6.14 unterschiedenen `compile`-Optionen ist festzustellen, daß die schnellsten Ergebnisse (über 12MHz) ohne `incremental`-Option entstehen und daß mit `min_paths` ähnliche Ergebnisplätze erzielt werden, wie ohne

Optionen. Dies entspricht der schon bei der High-Level-Synthese beobachteten Verteilung der Ergebnisse aufgrund der `compile`-Optionen.

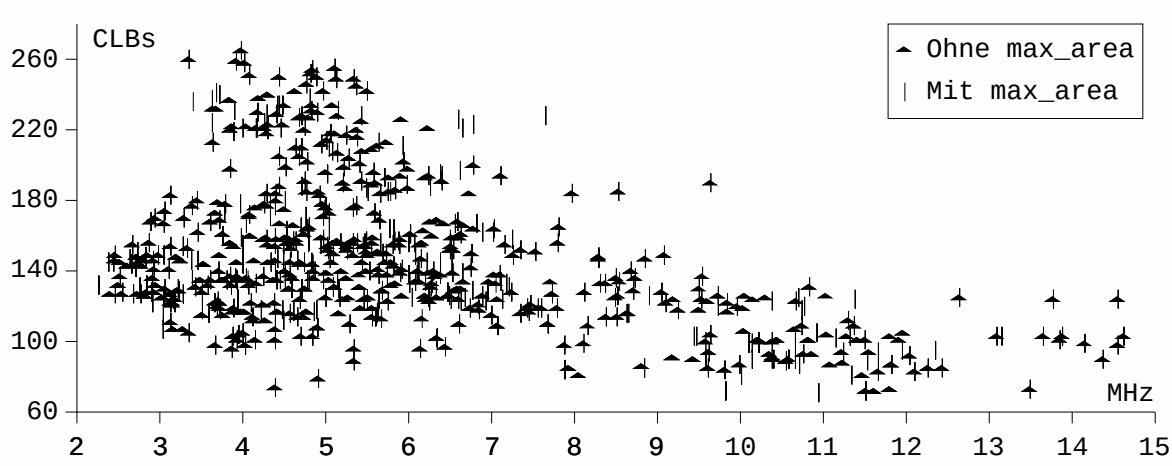


Bild 6.13: `set_max_area` nach der Controllersynthese des discount

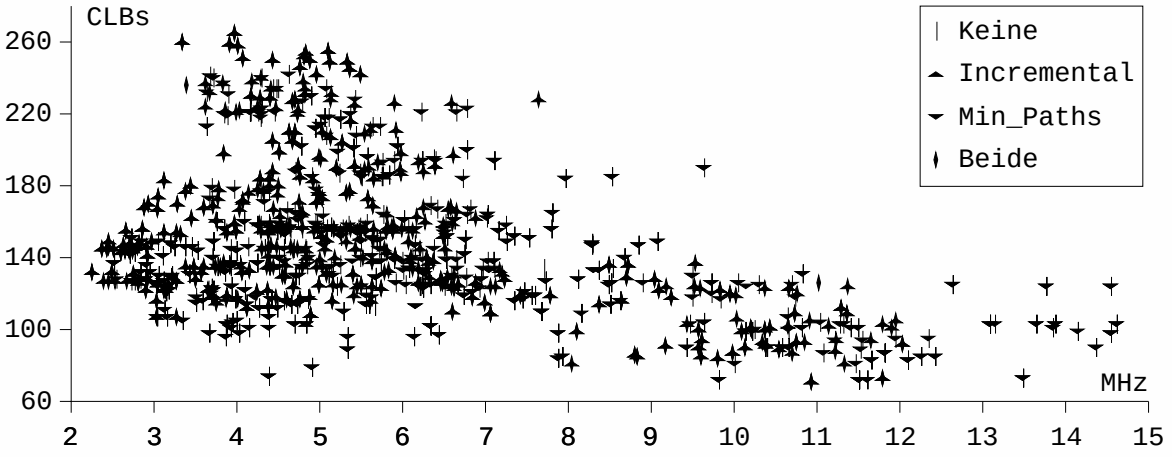


Bild 6.14: `compile`-Optionen nach der Controllersynthese des discount

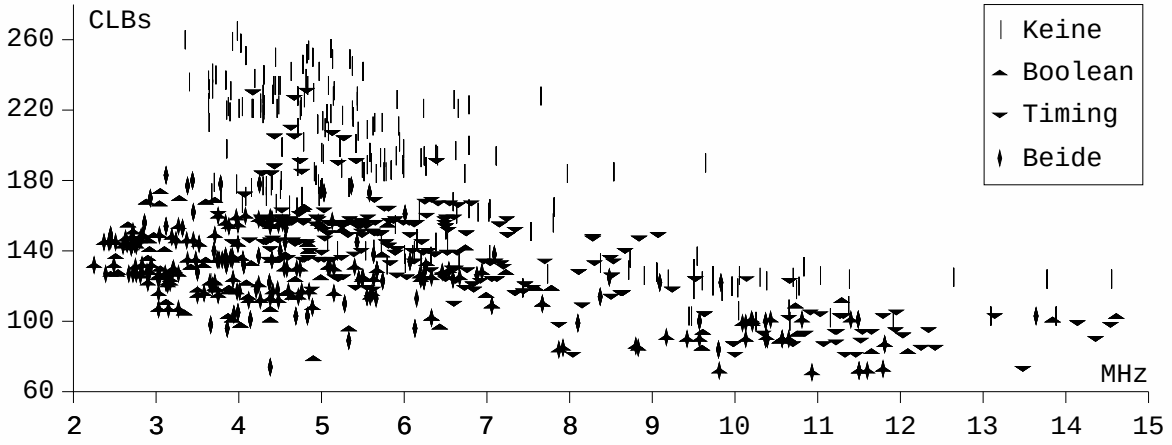


Bild 6.15: Strukturoptimierungen nach der Controllersynthese des discount

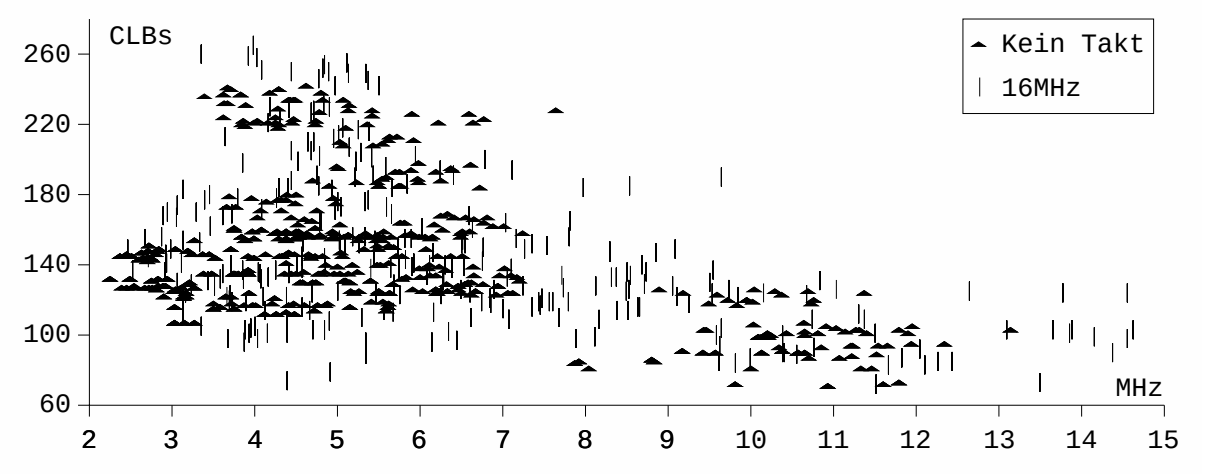


Bild 6.16: Taktvorgaben nach der Controllersynthese des discount

Ähnlich der direkten RTL-Synthese und der High-Level-Synthese entstehen große Ergebnisse schwerpunktmäßig ohne Strukturoptimierungen (Bild 6.15), wobei für die Schaltungsgeschwindigkeit keine Abhängigkeit erkennbar ist. Sehr schnelle Controller haben mit der direkten RTL-Synthese zudem gemeinsam, daß sie entsprechend Bild 6.16 eine Taktvorgabe besitzen. Der Umkehrschluß gilt jedoch nicht, da trotz einer Taktvorgabe langsame Controller entstehen können.

Abschlußbetrachtung des Bildschirmcontrollers: Die Projektion der Ergebnisse aller drei Synthesemethoden für ein 3090A-7-FPGA in ein einziges Diagramm (Bild 6.17) zeigt das Verhältnis der einzelnen Ergebnisbereiche zueinander. Die Modellierung in RTL-Verilog erzielt die mit deutlichem Abstand kleinsten und schnellsten Ergebnisse, gefolgt von der Controllersynthese. Die Lösungen der High-Level-Synthese liegen innerhalb des Bereiches der Controllersynthese.

Unter Einbeziehung der Entwurfszeit ist damit die High-Level-Synthese die für den Bildschirmcontroller schlechteste Synthesemethode, was auf das im Taktraster fest vorgegebene I/O-Protokoll und die wenigen, einfachen Operationen zurückzuführen ist [Friedr98]. Die High-Level-Optimierungsverfahren und die algorithmische Beschreibungsmethode sind bei diesem Entwurf nicht nutzbar.

Bei der Abwägung zwischen direkter RTL-Synthese und Controllersynthese sind die zur Verfügung stehende Entwurfszeit und die Anforderungen an die Lösung entscheidend. Ist die Entwurfszeit kritisch und kann eine Zieltechnologie so gewählt werden, daß eine ausreichend schnelle Schaltung entsteht, so ist die Controllersynthese zu bevorzugen. Steht dagegen die Zieltechnologie fest und sind die Anforderungen an Größe und Geschwindigkeit kritisch, so ist trotz ihrer höheren Entwurfsdauer die direkte RTL-Synthese im Vorteil, da ansonsten keine verwertbare Lösung erreicht werden kann.

Zusätzlich zu dem dominierenden Einfluß der Modellierung ist bei diesem Entwurf übergreifend über alle drei Methoden erkennbar:

- Taktvorgaben sind Voraussetzung für besonders schnelle Ergebnisse
- Sehr kleine Schaltungen entstehen nur bei Strukturoptimierungen

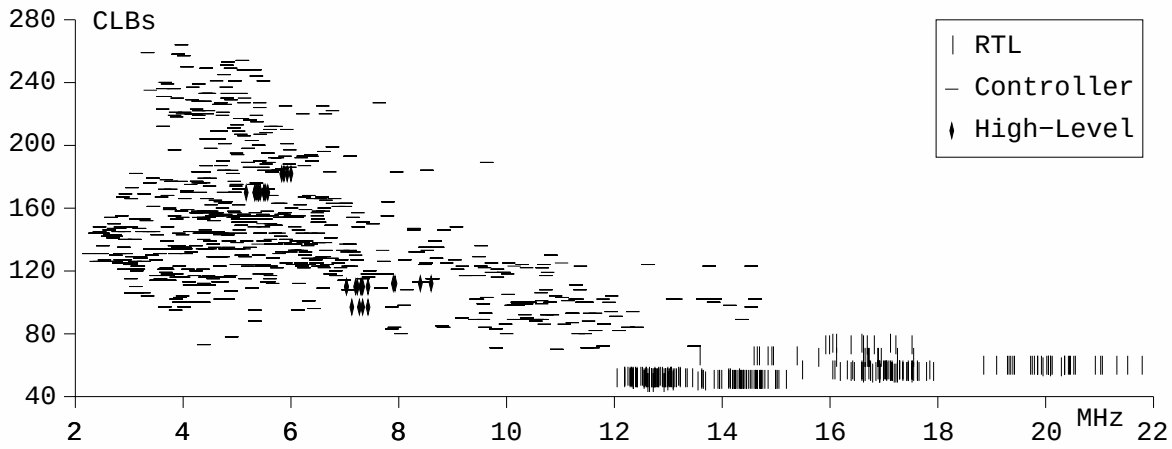


Bild 6.17: Ergebnisse der drei Synthesemethoden auf einem 3090A-7-FPGA

6.2.2 Entwurf eines Digital-Audio-Receivers

Mit allen drei Synthesemethoden behandelt wurde auch der Entwurf des Digital-Audio-Receivers (DAR), einem Dekoder für digitale Audiodaten [SonPhi83]. Die Schaltung wurde für ein VLSI-Praktikum entwickelt [Blinze97] und in [Friedr98] mit RTL-Synthese und High-Level-Synthese implementiert. Bei den Beta-Tests des Protocol-Compilers erfolgte die Controllermodellierung und -synthese, deren erfolgreiches Ergebnis auf der Design Automation Conference 1997 [HolBli98] vorgestellt und praktisch demonstriert wurde. Die Schaltung hatte für jede der verwendeten Synthesemethoden folgende Anforderungen zu erfüllen:

- Abtastung des digitalen Eingangssignals mit 16MHz, dem Eingangstakt
- Serielles Kommunikationsprotokoll zum Digital-Analog-Converter (DAC)
- Implementierung auf möglichst kleinem und langsamen Xilinx-3000-FPGA
- Synchronisation der Verarbeitung mit Datentakt durch digitale PLL
- Erkennung und Unterdrückung ungültiger oder fehlerhafter Eingangsdaten

Aufgrund der unterschiedlichen Ergebnisse für die Synthesemethoden erfolgte wie beim Bildschirmcontroller zunächst die Implementierung auf unterschiedlich großen und schnellen FPGAs. Für den abschließenden Vergleich der Methoden wurde als einheitliches Ziel-FPGA der Typ 3190A-3 benutzt.

RTL-Synthese des DAR: Hierbei wurde die Musterlösung der Praktikumsaufgabe [Blinze97] der alternativen Lösung aus [Friedr98] gegenübergestellt. Die Modelle lösen die Struktur feinkörnig in Register und kombinatorische Logik auf und erforderten eine Entwurfszeit von etwa einer Woche.

Es wurden die Taktvorgabe, die Flächenvorgabe, die Strukturoptimierungen und die Übersetzungsoptionen mit unterschiedlichen Einstellungen untersucht. Dabei ergab sich für den FPGA-Typ 3042-125 ein weit gestreutes Ergebnisspektrum für die beiden Modelle, das in Bild 6.18 dargestellt und in Tabelle A.6 aufgelistet ist. Ähnlich dem Bildschirmcontroller besteht eine Aufteilung der

Ergebnisse in zwei getrennte Teilbereiche, die auch hier durch das unterschiedliche Verhältnis von Registern und kombinatorischer Logik der Modelle entsteht.

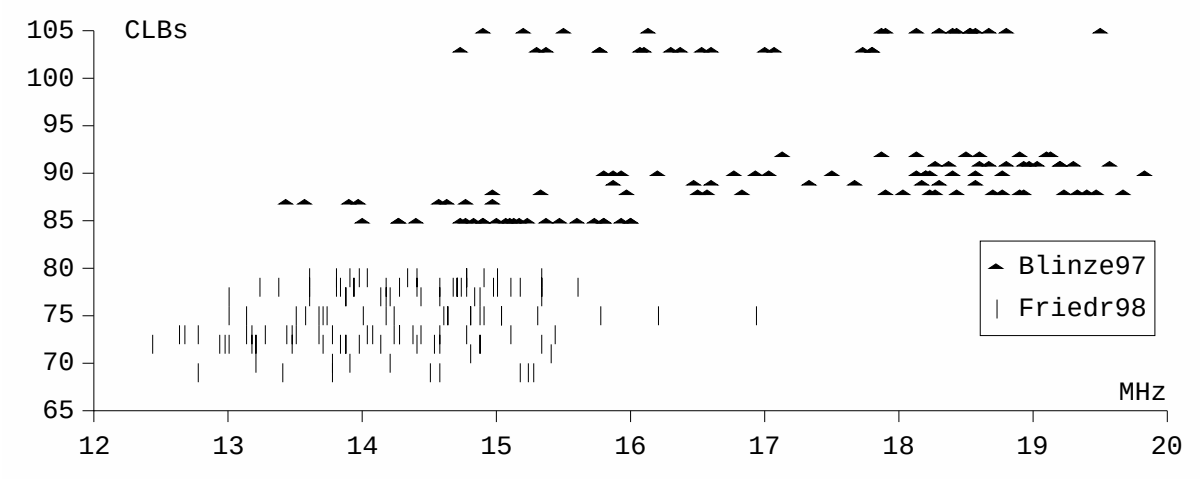


Bild 6.18: RTL-Modellierungsvarianten des DAR

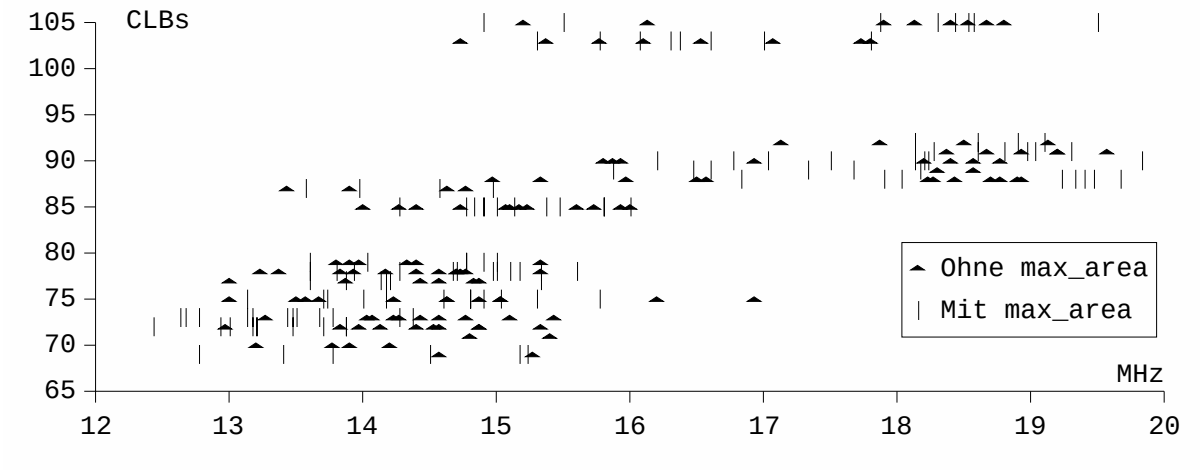


Bild 6.19: `set_max_area` bei der RTL-Synthese des DAR

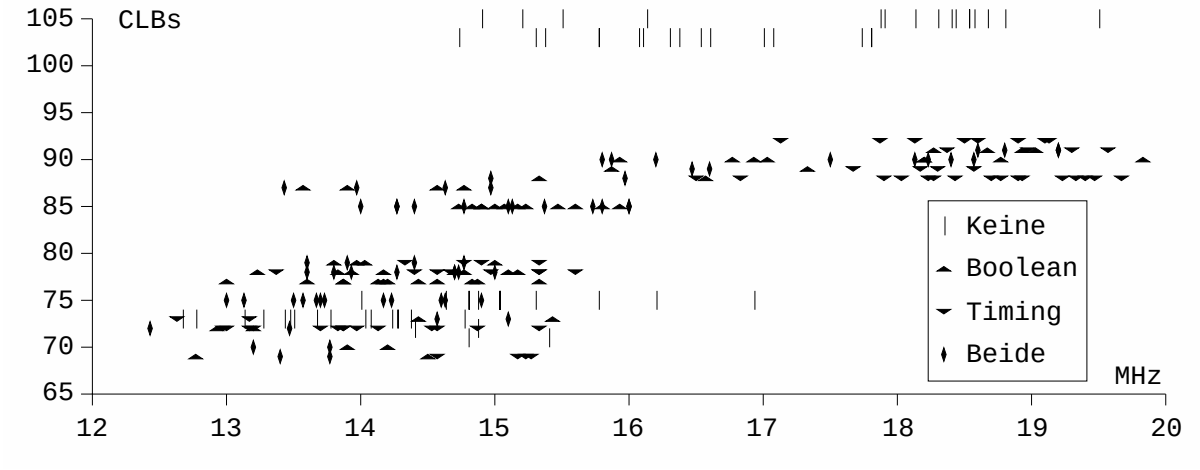


Bild 6.20: Strukturoptimierungen bei der RTL-Synthese des DAR

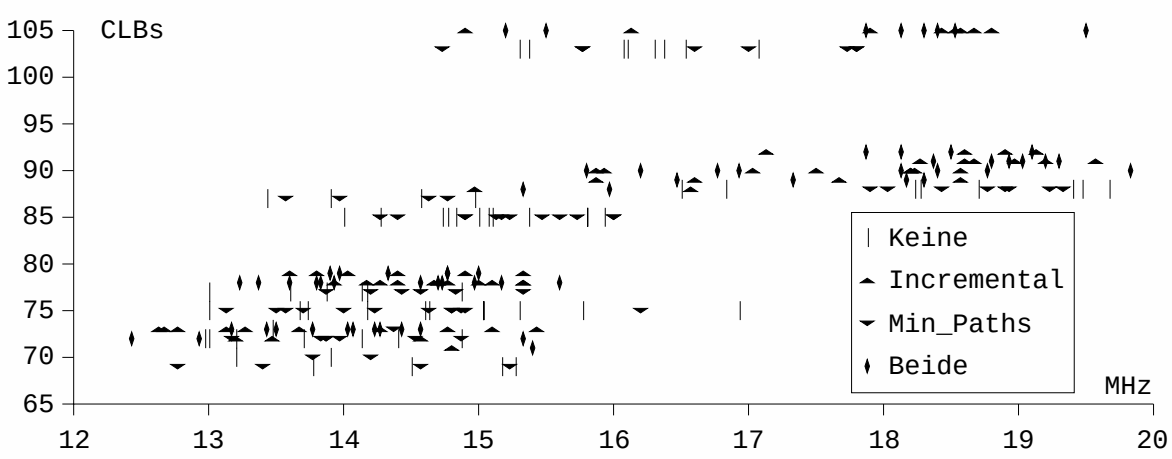


Bild 6.21: `compile`-Optionen bei der RTL-Synthese des DAR

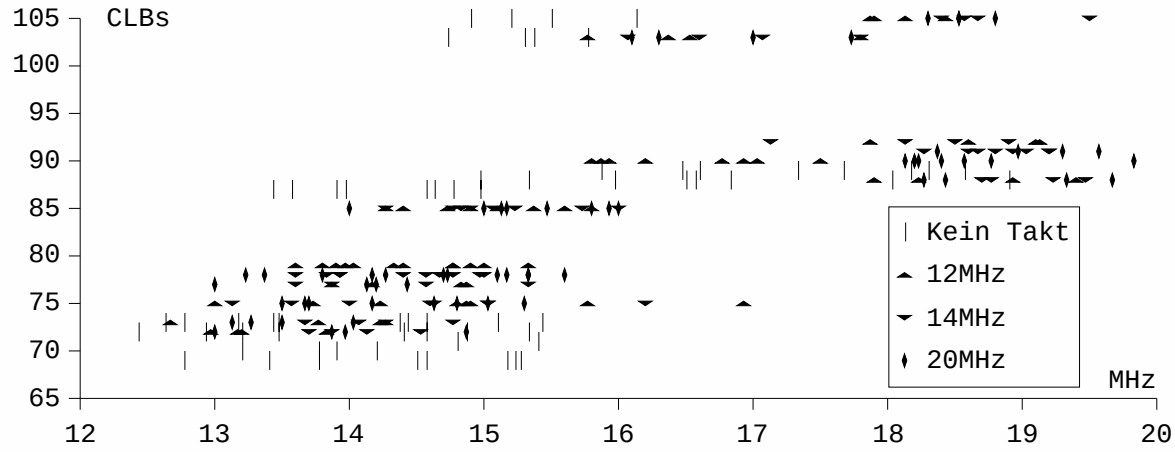


Bild 6.22: Taktvorgaben bei der RTL-Synthese des DAR

Die Kennzeichnung der Ergebnisse nach verwendeten Syntheseoptionen für die Flächenvorgabe (Bild 6.19), die Strukturoptimierungen (Bild 6.20), die `compile`-Optionen (Bild 6.21) und die Taktvorgaben (Bild 6.22) bestätigt im wesentlichen die beim Bildschirmcontroller beobachteten Abhängigkeiten. Die Flächenvorgabe und die `compile`-Optionen besitzen auch beim DAR keinen gezielten Einfluß und hohe Taktraten werden schwerpunktmäßig bei Taktvorgaben erreicht. Die kleinsten Schaltungen entstehen wiederum mit Strukturoptimierungen, jedoch sind beim DAR anders als beim Bildschirmcontroller für diese Optimierungen keine klaren Einflüsse auf die Geschwindigkeit erkennbar.

High-Level-Synthese des DAR: Das hierbei verwendete Modell wurde im Rahmen von [Friedr98] in ungefähr einer Woche entwickelt. Da in diesem Modell keine Operator- oder Ressourcentypen mehrfach benutzbar sind, bietet es keine Ansätze für High-Level-Entwurfsoptimierungen durch Scheduling und Allocation. Ähnlich dem Bildschirmcontroller ist die High-Level-Synthese auf die Konstruktion von FSMD-Strukturen mit geeignetem I/O-Timing für die vier Teilmodule beschränkt,

wobei ein zeitunkritischer 5-Bit Inkrementer die einzige komplexe Operation ist. Die bei dieser Methode untersuchten Variationen beschränkten sich daher auf die abschließende RTL-Synthese, bei der unterschiedliche Vorgaben für die Fläche, die Strukturoptimierungen und die `compile`-Optionen erfolgten. Die nachfolgend graphisch betrachteten Ergebnisse dieser Untersuchungen auf einem 3190A-3-FPGA sind in Tabelle A.7 aufgeführt.

Die in Bild 6.23 vorgenommene Markierung der Ergebnisse nach `compile`-Optionen weist wie bereits beim Bildschirmcontroller auf eine Unverträglichkeit von High-Level-Synthese und `incremental_map` hin, da auch hier mit dieser Option erzielte Ergebnisse signifikant größer und langsamer sind als die übrigen. Diese Ergebnisse werden daher nicht weiter berücksichtigt. Für die `min_paths`-Option ergibt sich ebenso wie für die in Bild 6.24 gekennzeichnete Flächenvorgabe wiederum keine zielgerichtete Wirkung. Die Strukturoptimierungen zeigen als einzige Option Abweichungen zum Bildschirmcontroller (Bild 6.25), da sie hier die kleinsten und die schnellsten Ergebnisse erzeugen, wobei eine reine Timing-Optimierung wieder die kleinsten Ergebnisse liefert.

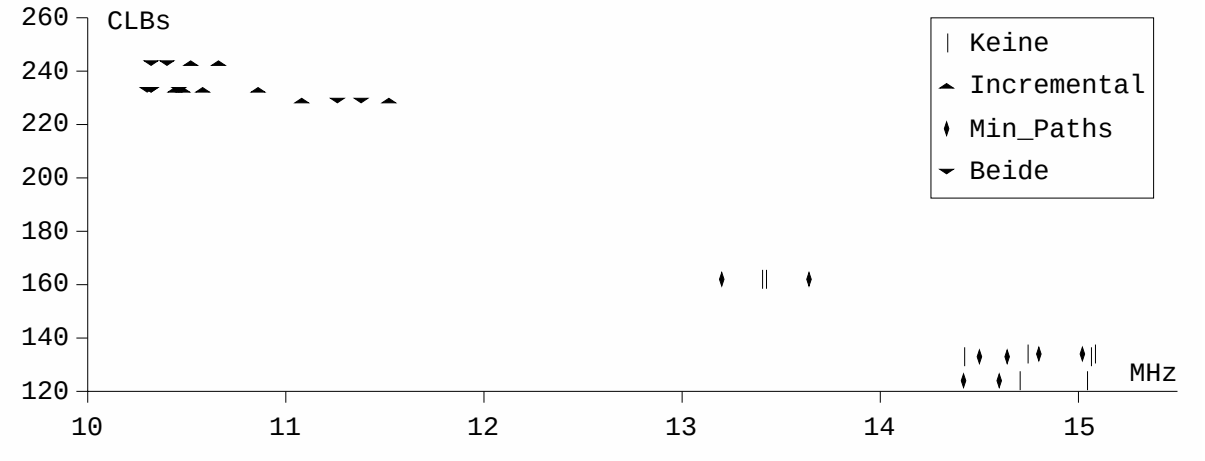


Bild 6.23: `compile`-Optionen nach der High-Level-Synthese des DAR

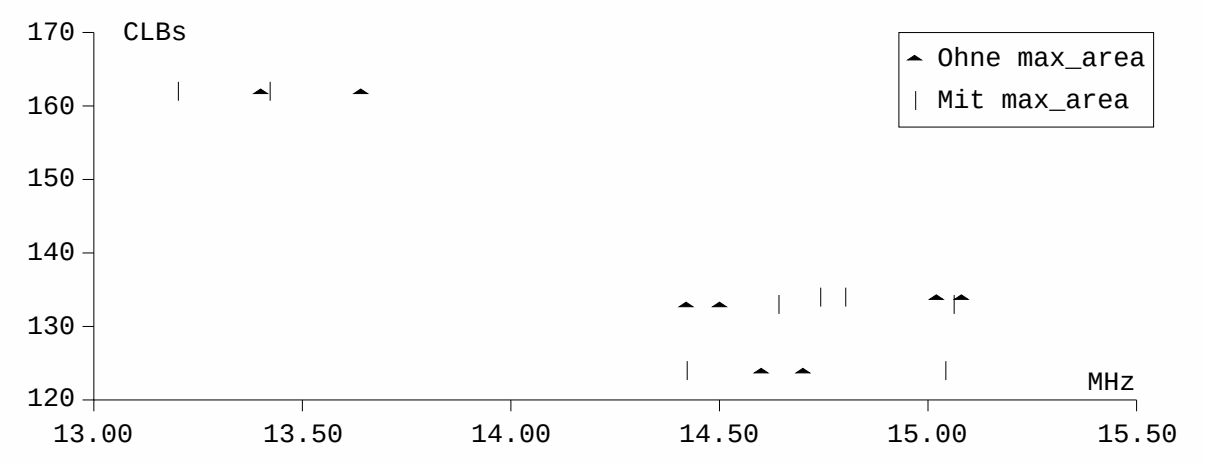


Bild 6.24: `set_max_area` nach der High-Level-Synthese des DAR

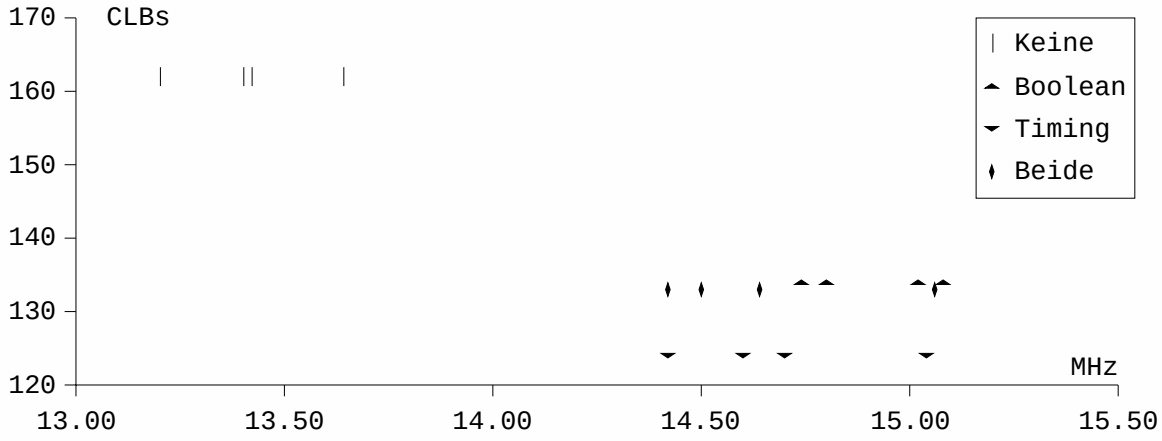


Bild 6.25: Strukturoptimierungen nach der High-Level-Synthese des DAR

Controllersynthese des DAR: Als Grundlage dieser Untersuchungen wurde das in [HolBli98] vorgestellte Modell benutzt, das sich aus vier parallel arbeitenden Teilbeschreibungen zusammensetzt. Die Entwurfszeit dieses Modells betrug etwa eine halbe Woche. Hierbei wurden die Codierungen und Prozeßarchitekturen der Controllersynthese sowie die Optionen der anschließenden RTL-Synthese variiert. Die dabei erzielten Ergebnisse für den FPGA-Typ 3064A-7 zeigt Tabelle A.8.

Das vollständige Modell konnte global nur als eine Distributed-Partition ohne Zustandsanalyse codiert werden. Die übrigen Codierungen und die automatische Partitionierung setzen eine vollständige Zustandsanalyse voraus, die bei diesem Entwurf undurchführbar war. Durch eine manuelle Partitionierung, die testweise mit einer Binary-Min-Code für das Frame Send_to_DAC und einer Distributed-Code für die anderen Teile untersucht wurde, sind weitere Codierungen möglich. Diese erfordern nicht automatisierbare Eingriffe in das Modell bei exponentiell mit der Anzahl betrachteter Partitionen wachsenden Codierungsmöglichkeiten und sind damit bei diesem Entwurf nicht umfassend untersuchbar. Die Ergebnisse der zwei Codierungsvarianten sind in Bild 6.26 gegenübergestellt und zeigen, daß die manuelle Partitionierung in diesem Fall kompaktere Ergebnisse liefert als der globale Distributed-Code. Bei platzkritischen Entwürfen, für die eine Zustandsanalyse nicht möglich ist, ist eine manuelle Partitionierung somit zu erwägen.

Bei der Unterscheidung der Ergebnisse nach Prozeßarchitekturen des Verilog-Modells in Bild 6.27 liefern die Split- und die Multi-Process-Architecture die kompaktesten Ergebnisse. Im Vergleich zum Bildschirmcontroller wirkt sich die Prozeßarchitektur auf die Schaltungsgröße also in umgekehrter Weise aus.

Bei den Variationen der RTL-Syntheseeinstellungen werden für die Flächenvorgabe wie bei den anderen Methoden und Entwürfen keine Abhängigkeiten deutlich (Bild 6.28). Die in Bild 6.29 hervorgehobenen `compile`-Optionen lassen ebenfalls keine gezielten Einflüsse erkennen, wobei sich die `min_paths`-Option überhaupt nicht auswirkt. Die kleinsten Ergebnisse entstehen wie sonst auch mit Strukturoptimierungen (Bild 6.30).

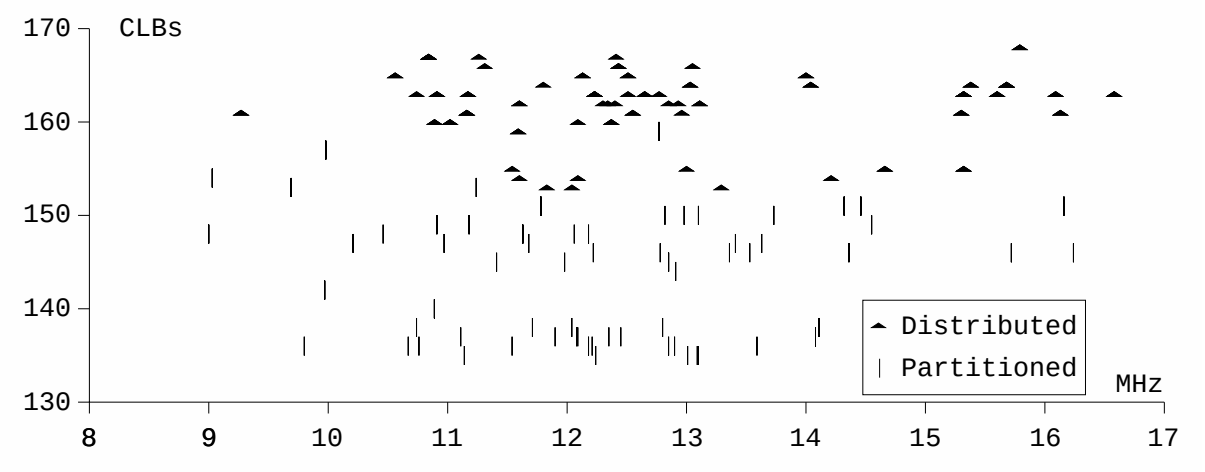


Bild 6.26: Codierungsvarianten der DAR-Controllersynthese

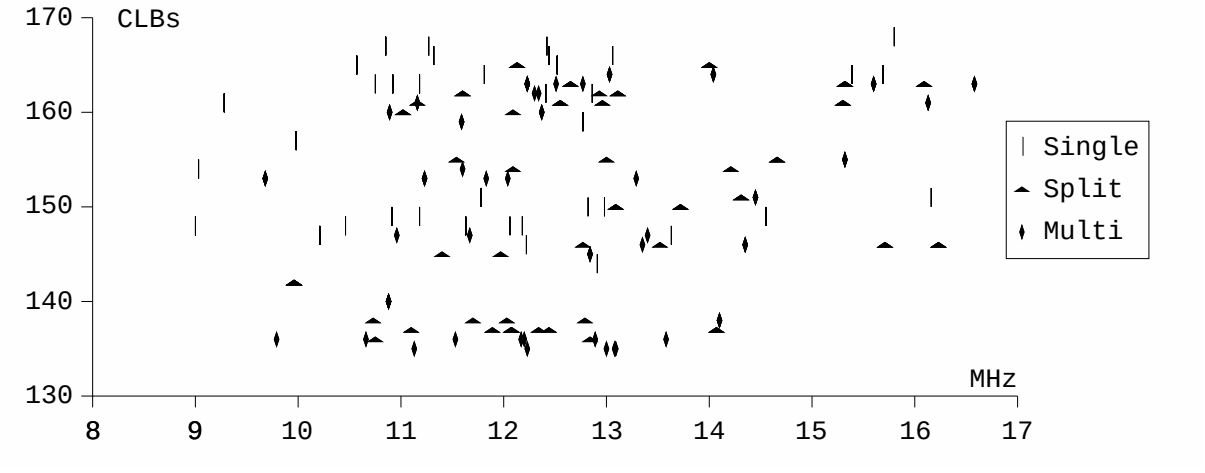


Bild 6.27: Modellstrukturen der DAR-Controllersynthese

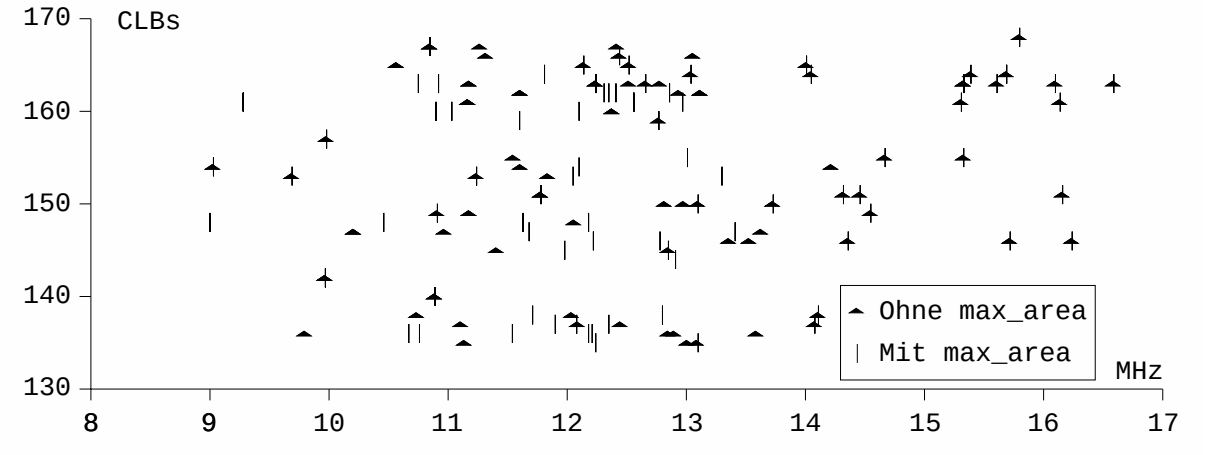


Bild 6.28: `set_max_area` nach der Controllersynthese des DAR

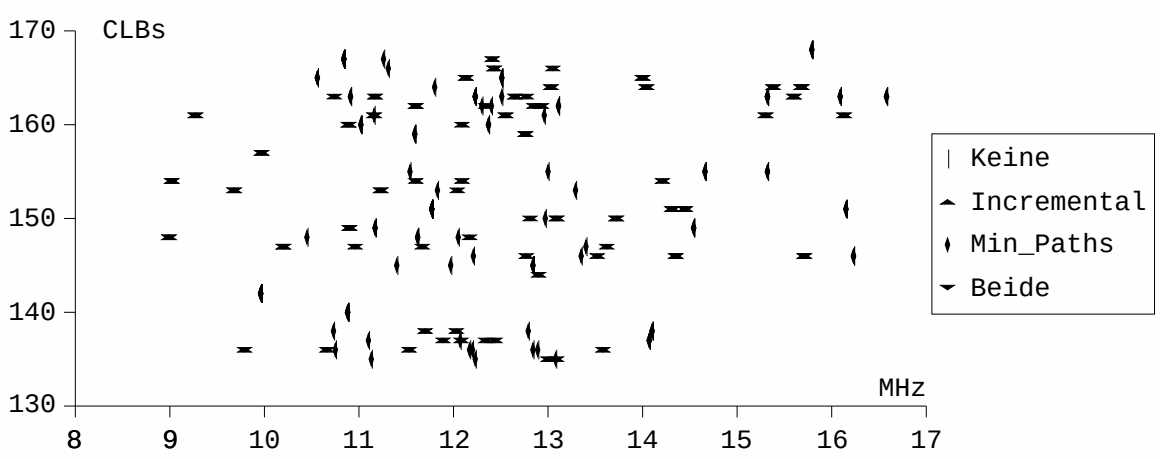


Bild 6.29: `compile`-Optionen nach der Controllersynthese des DAR

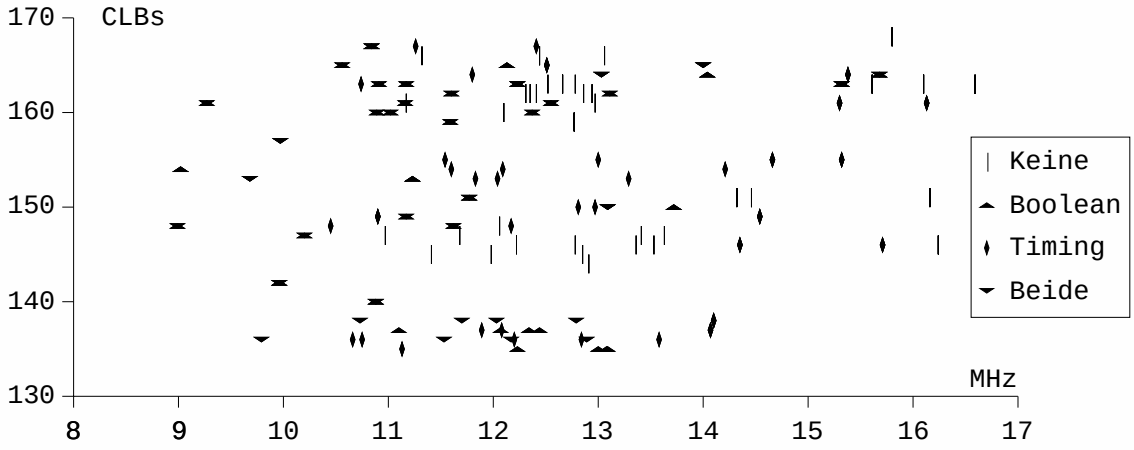


Bild 6.30: Strukturoptimierungen nach der Controllersynthese des DAR

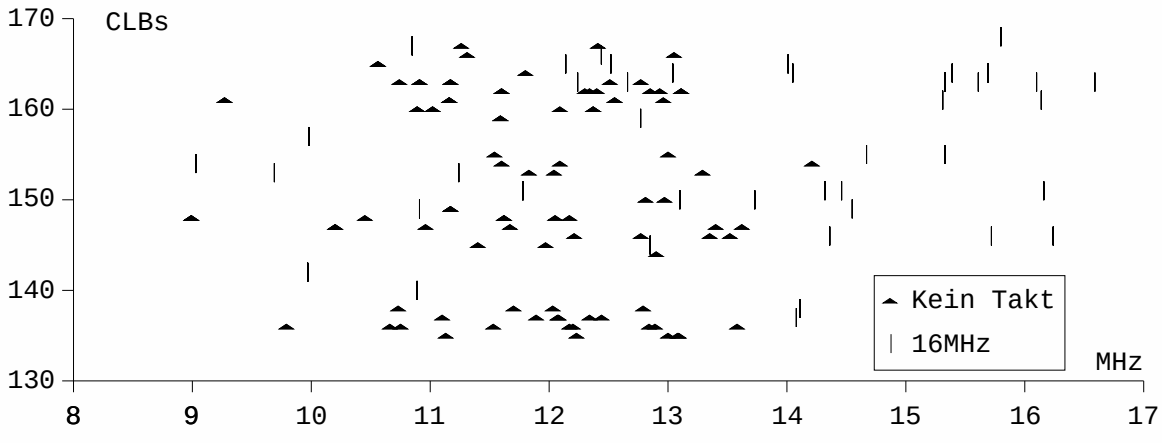


Bild 6.31: Taktvorgaben nach der Controllersynthese des DAR

Bei der Kennzeichnung der Taktvorgaben in Bild 6.31 bestätigt sich wiederum die Beobachtung, daß bei der RTL-Synthese Taktvorgaben eine Voraussetzung für besonders schnelle Ergebnisse sind.

Zusammenfassung der DAR-Synthese: Die Darstellung aller Ergebnisse der drei Synthesemethoden des DAR in Bild 6.32 auf einem einheitlichen Ziel-FPGA des Typs 3190A-3 zeigt im Vergleich zum Bildschirmcontroller Ähnlichkeiten in Form und Verhältnis der Ergebnisbereiche von direkter RTL-Synthese und High-Level-Synthese. Der Ergebnisbereich der Controllersynthese fällt hier jedoch sehr klein aus, was auf die wenigen verwendbaren Codierungsmöglichkeiten des Controllers zurückzuführen ist.

Die Modellierung in RTL-Verilog erzielt bei ähnlicher Entwurfszeit deutlich kleinere und schnellere Ergebnisse, als eine High-Level-Beschreibung. Dies ist auf die wenigen einfachen Operationen zurückzuführen, die keinen Ansatz für High-Level-Optimierungen bieten. Zwar führt die algorithmische Beschreibung zu einem kürzeren und verständlicheren Modell [Friedr98], insgesamt ist die High-Level-Synthese aber für diesen Entwurf vergleichsweise ungeeignet.

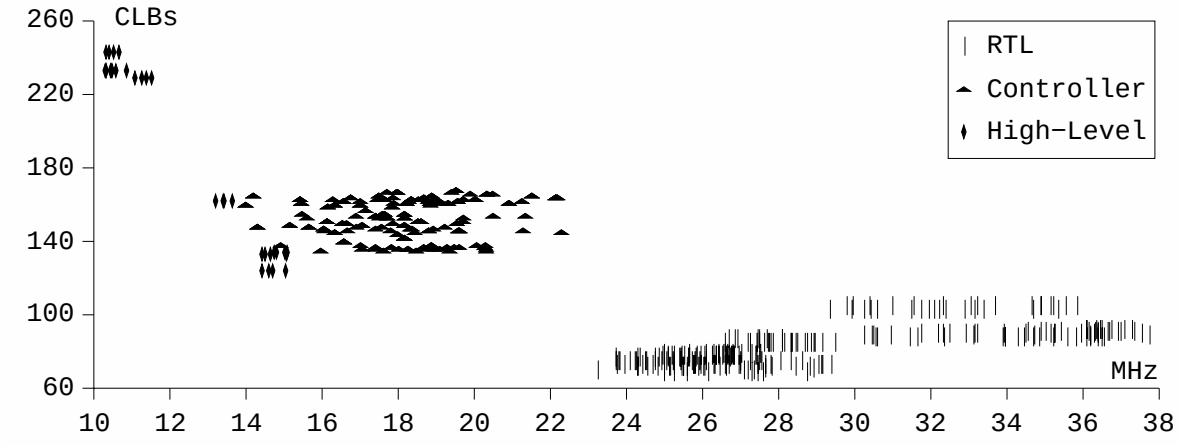


Bild 6.32: Ergebnisse der DAR-Synthesemethoden auf einem 3190A-3-FPGA

Bei der Entscheidung zwischen RTL-Synthese und Controllersynthese sind die zur Verfügung stehende Entwurfszeit und die Anforderungen an die Lösung wie schon beim Bildschirmcontroller entscheidend. Soll der Entwurf in möglichst kurzer Zeit erfolgen und kann die Zieltechnologie an dessen Ergebnis angepaßt werden, so ist die Controllersynthese im Vorteil. Sind dagegen kritische Grenzen für Größe und Laufzeiten vorgegeben, so wäre die Modellierung in RTL-Verilog zu bevorzugen.

Bei der Controllersynthese des DAR wurde zudem deutlich, daß durch die Art der Modellierung der Zeitbedarf für die Zustandsanalyse und -optimierung bis zur praktischen Undurchführbarkeit erhöht werden kann. Es darf also nicht darauf vertraut werden, daß für eine beliebige Modellierung ein Optimierungsspielraum durch Codierungsvarianten vorhanden ist.

Übereinstimmend mit den Beobachtungen für den Bildschirmcontroller sind auch beim DAR übergreifend über alle drei Methoden neben dem dominanten Einfluß der Modellierung als Eigenschaften der RTL-Syntheseoptionen erkennbar:

- Taktvorgaben sind Voraussetzung für besonders schnelle Ergebnisse
- Sehr kleine Schaltungen entstehen nur bei Strukturoptimierungen
- Die Auswirkungen einer Flächenvorgabe sind nicht zielgerichtet
- Die `compile`-Optionen verschlechtern Ergebnisse eher, als daß sie diese verbessern

6.2.3 Entwurf eines einfachen RISC-Prozessors

Eine Erweiterung des in [ScWaTe94] beschriebenen URISC-Mikroprozessors um ALU-Operationen und Pointer-Register für indirekte Speicheradressierung wurde unter dem Namen Move-RISC-Prozessor (MRISC) in einem Entwurfspraktikum entwickelt und in dieser Arbeit mittels RTL-Synthese und High-Level-Synthese untersucht.

Als Modelle dienten die Musterlösung der Praktikumsaufgabe [Blinze96] und die Lösungen aus [Friedr98] in RTL- bzw. High-Level-Verilog. An die Entwürfe wurden im Rahmen der Syntheseexperimente diese Anforderungen gestellt:

- Xilinx-FPGA 4010XLPC84-3 als Zielhardware
- Erreichen einer möglichst hohen Taktrate
- Fest vorgegebenes, zyklisches I/O-Schema mit 4 Takten je Instruktionszyklus
- Operationen im Prozessor abhängig von Quell- und Zieladresse des Move
- Verarbeitung von 16-Bit Daten und Adressen
- Arithmetische und logische Verknüpfungen von Daten
- Bidirektionaler Datenbus mit synchronem Busprotokoll

Im Vergleich zu den kontrollflußdominierten Entwürfen discount und DAR, die zu einem wesentlichen Teil interne Zustände für die Berechnung von Ausgaben und Folgezuständen benutzen, wird der MRISC weitgehend durch die Daten aus dem Programmspeicher gesteuert. Lediglich die Speicherzugriffe erfolgen zyklisch nach einem festgelegten Kontrollschema aus vier Takten. Durch die 16-Bit breite Adress- und Datenverarbeitung kommen beim MRISC zudem reguläre logische und arithmetische Verknüpfungen zur Anwendung.

RTL-Synthese des MRISC: Die betrachteten RTL-Verilog-Modelle [Blinze96] und [Friedr98], deren Entwurfszeit bei jeweils etwa einer Woche lag, sind gemäß der Aufgabenstellung in ein Hauptmodul und drei Untermodule aufgeteilt, verwenden innerhalb der Module aber verschiedene Aufteilungen in `always`-Prozesse, Register und kombinatorische Logiken.

Wie bei den anderen RTL-Syntheseexperimenten wurden auch diese Modelle mit Variationen der Einstellungen für Flächenvorgabe, Strukturoptimierungen, `compile`-Optionen und Taktvorgaben untersucht. Außerdem wurde neben der globalen, hierarchiefreien Synthese die modulweise Übersetzung der Entwurfshierarchie in Bottom-Up-Richtung durchgeführt. Die in Bild 6.33 unterschiedenen Modelle überschneiden sich in ihren jeweiligen Ergebnisbereichen weiträumig, was auf eine geringere Bedeutung der Modellunterschiede in Prozeßaufteilung, Registern und kombinatorischer Logik gegenüber den Gemeinsamkeiten in den

verwendeten Operationen hinweist. Die dargestellten Daten sind in Tabelle A.10 zusammengefaßt.

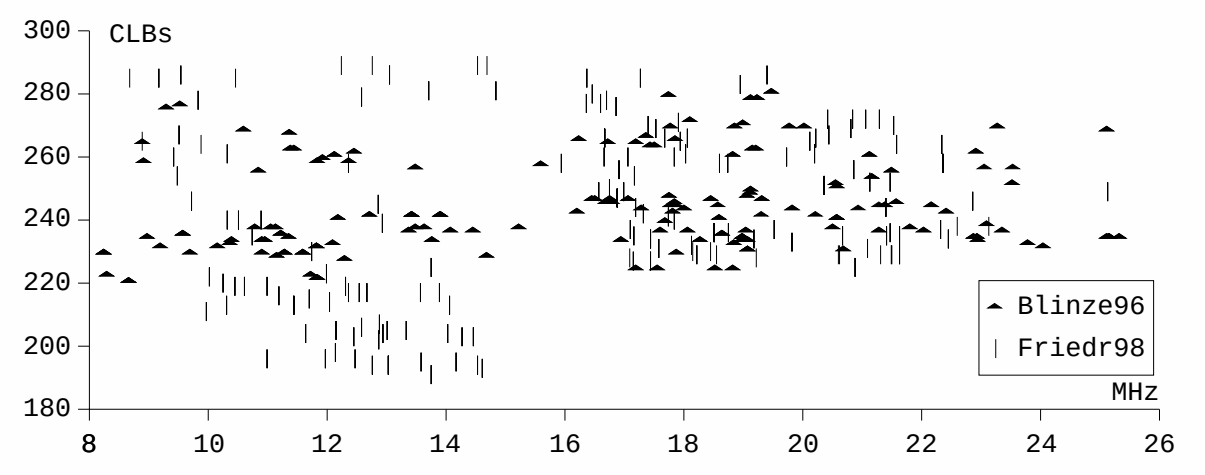


Bild 6.33: RTL-Modellierungsvarianten des MRISC

Eine Tendenz zu kleineren Ergebnissen ist bei der globalen Synthese gegenüber der modularen Synthese vorhanden (Bild 6.34), ohne jedoch die Geschwindigkeit zu beeinflussen. Die Aufhebung der Modulgrenzen bei der globalen Synthese bietet der Optimierung mehr Möglichkeiten zur Zusammenfassung und Vereinfachung von Logik. Die globale Synthese benötigt gegenüber der modularen Synthese aber auch mehr Rechenzeit und Rechnerspeicher zur Ausführung.

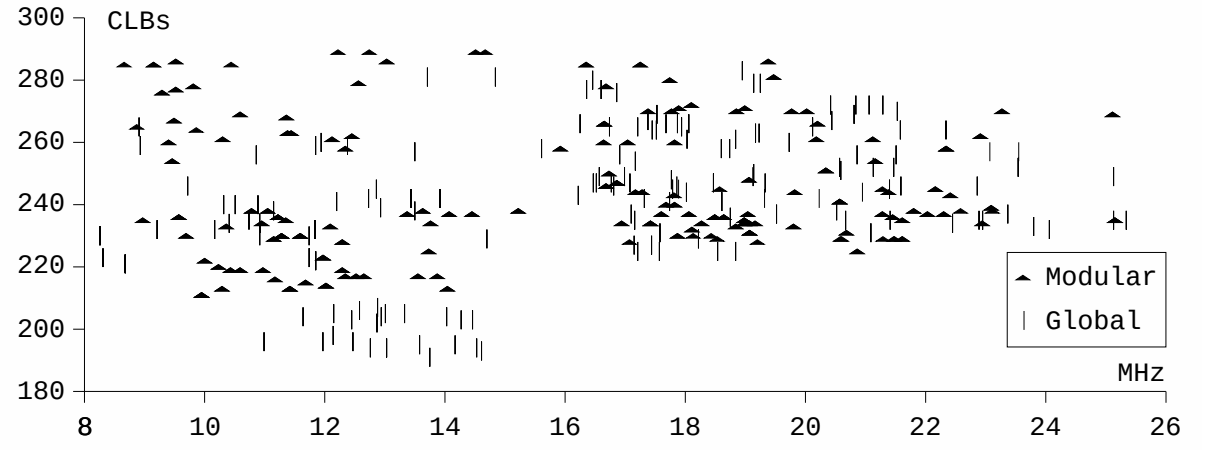


Bild 6.34: Modularisierung der RTL-Synthese des MRISC

Für die in Bild 6.35 hervorgehobene Flächenvorgabe ist wie bisher keine gerichtete Wirkung auf Größe oder Geschwindigkeit der Ergebnisse sichtbar. Bild 6.36 zeigt für die Strukturoptimierungen noch deutlicher als in anderen Versuchen die gewohnte Tendenz zu kleineren Schaltungen bei insgesamt neutraler Wirkung auf die Geschwindigkeit. Die Kennzeichnung der Übersetzungsoptionen in Bild 6.37 läßt zwar keine gerichtete Wirkung deutlich werden, die Überlagerung vieler Ergebnisse für keine Option und `min_paths` bzw. `incremental_map` und beide Optionen erlaubt aber die Folgerung, daß die `min_paths`-Option hier weitgehend bedeutungslos ist. Die Unterscheidung der Taktvorgaben (Bild 6.38) bestätigt die

Abhängigkeit zwischen Taktvorgaben und schnellen Ergebnissen und zeigt dabei eine klare Strukturierung des Ergebnisspielraums in getrennte Teilbereiche, die durch die Vorgabewerte bestimmt sind.

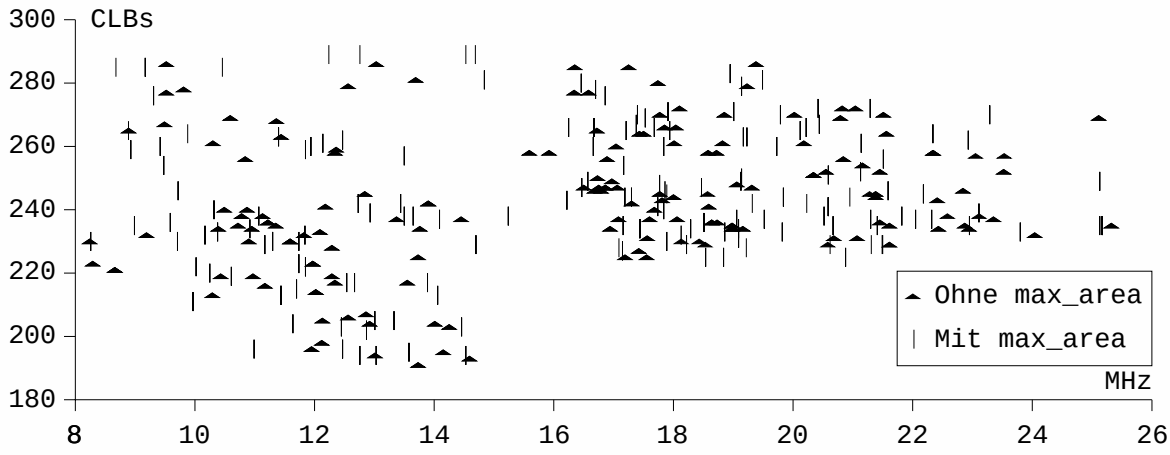


Bild 6.35: `set_max_area` bei der RTL-Synthese des MRISC

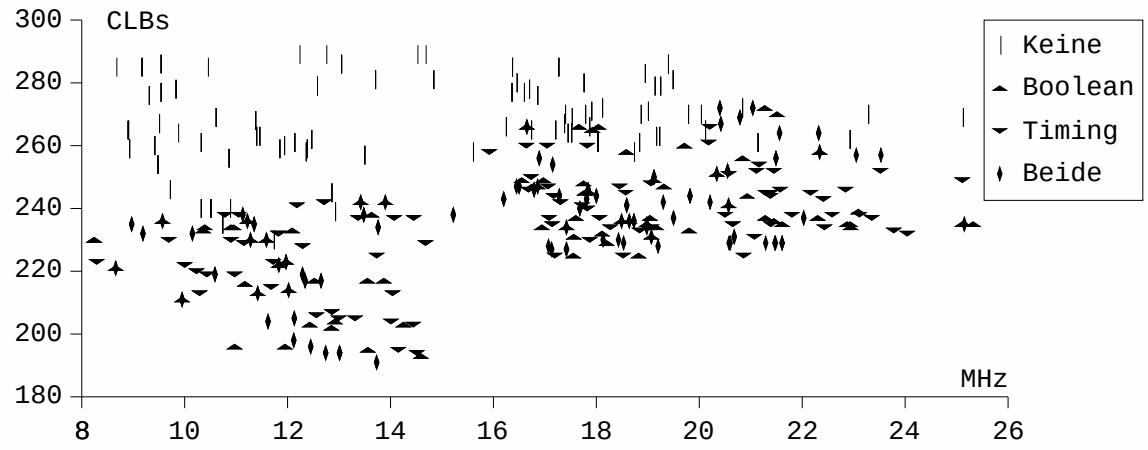


Bild 6.36: Strukturoptimierungen bei der RTL-Synthese des MRISC

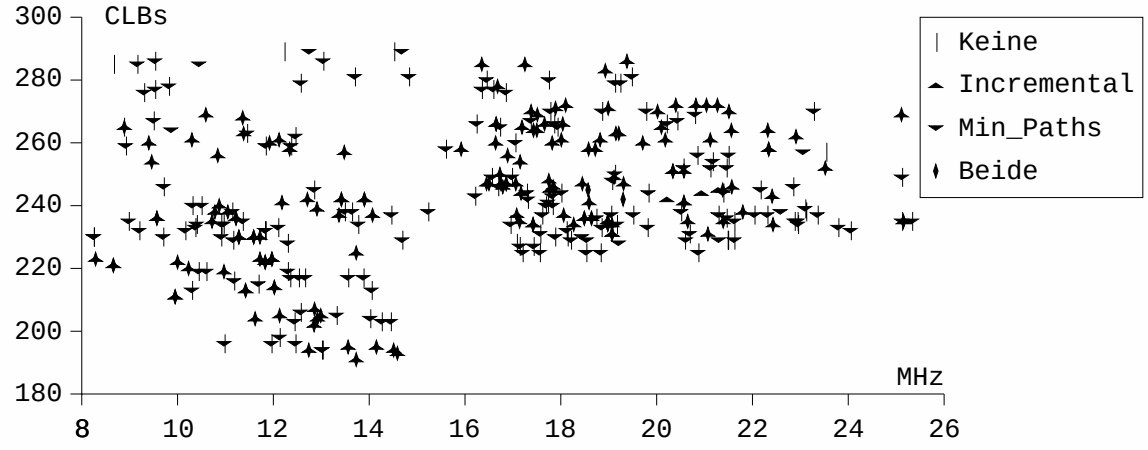


Bild 6.37: `compile`-Optionen bei der RTL-Synthese des MRISC

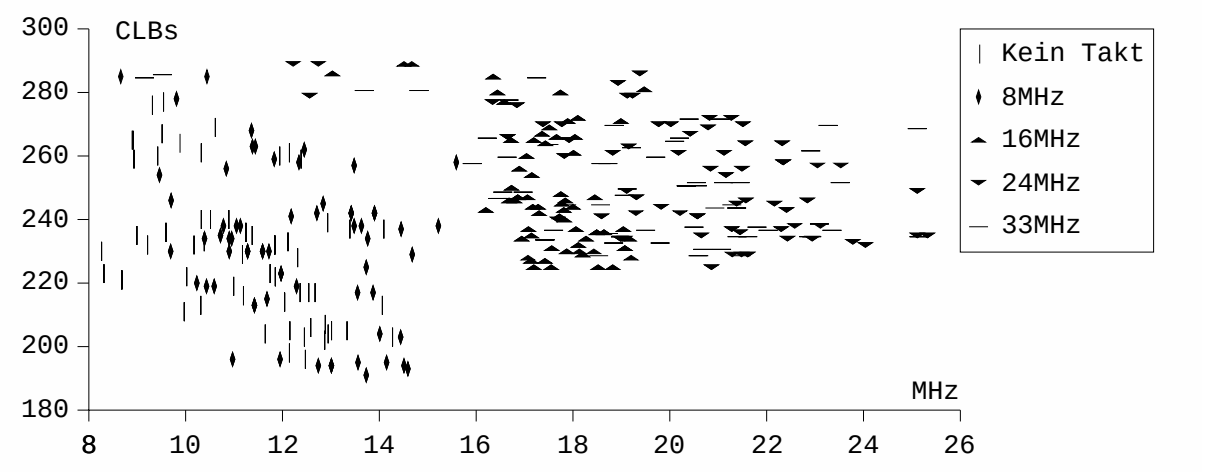


Bild 6.38: Taktvorgaben bei der RTL-Synthese des MRISC

Zur genaueren Untersuchung dieser Aufteilung wurde die Taktvorgabe in einem weiteren Syntheseexperiment von 0,2 bis 40MHz in Schritten zu 200kHz variiert. Dabei wurden eine minimale Flächenvorgabe, beide Strukturoptimierungen und keine `compile`-Optionen benutzt. Die aufgrund der Vorgabe resultierende Taktrate bzw. Schaltungsgröße sind in Tabelle A.11 aufgelistet und in Bild 6.39 bzw. Bild 6.40 dargestellt.

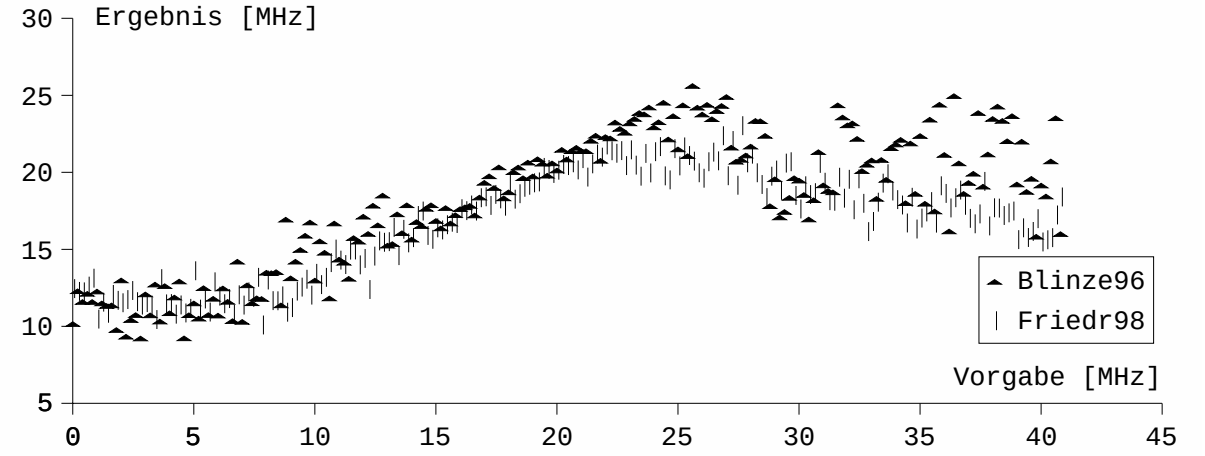


Bild 6.39: Einfluß von Taktvorgaben auf die Geschwindigkeit bei RTL-MRISC

Die Geschwindigkeit entwickelt sich von einem auch ohne Vorgabe erreichbaren Plateau an fast linear wachsend bis zu einem Maximum bei etwa 25MHz für den vorgegebenen und erzielten Wert. Die Forderung noch höherer Taktraten ergibt fast nur noch langsamere Ergebnisse mit Einbrüchen bis hinunter zu 15MHz, also 60% des möglichen Maximalwertes. Die Taktvorgabe darf also offensichtlich nicht überzogen werden, wenn maximale Schnelligkeit erreicht werden soll, sondern muß sich in der erreichbaren Zielregion bewegen. Damit ist ein iterativer Ansatz für die Bestimmung der Vorgabe unvermeidbar.

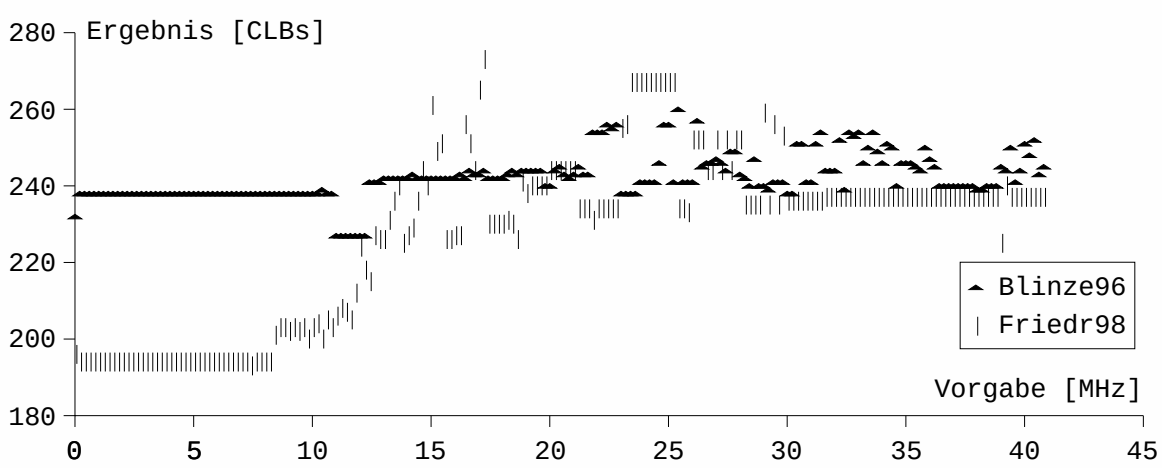


Bild 6.40: Einfluß von Taktvorgaben auf die Schaltungsgröße bei RTL-MRISC

Auf die Größe wirkt sich die Taktvorgabe für beide Entwürfe unterschiedlich aus, wobei es jeweils deutliche Sprungstellen gibt. Diese Sprünge ergeben sich durch einen Wechsel von Operator-Konstruktionen, z.B. von Ripple-Carry auf Carry-Look-Ahead-Addierer. Die laufzeitgesteuerte Plazierung und Verdrahtung glättet solche Sprünge im Geschwindigkeitsbereich, so daß sie dort nicht sichtbar werden.

Interessanterweise haben kleine Änderungen in der Taktvorgabe stellenweise deutliche Auswirken auf die Geschwindigkeit und Größe einer Schaltung. Die Variation der Taktvorgaben im angestrebten Zielbereich hilft also, offensichtlich vorhandene, lokale Optimierungsminima zu überwinden.

High-Level-Synthese des MRISC: Der High-Level-MRISC aus [Friedr98] besitzt eine grundlegend andere Struktur als die RTL-Modelle. Seine Funktionalität wird im wesentlichen in einem einzigen Modul mit nur einem algorithmischen Prozeß beschrieben, während zwei RTL-Module für die Zusammenführung der getrennten Lese- und Schreibdatenbusse des High-Level-Moduls zu einem bidirektionalen Bus bzw. die Kopplung des Kernmoduls mit dem Busmodul und die Schnittstellen nach außen sorgen. Die Modellierung des MRISC in einem sequentiellen Prozeß führte zu einer sehr kurzen Beschreibungsform und ersparte viel Arbeit bei der Beschreibung der internen Prozessorkommunikation. Dies schlug sich in einer Entwurfszeit von knapp einer halben Woche nieder.

Zusätzlich zu den RTL-Syntheseoptionen war bei diesem High-Level-Entwurf auch eine Variation der Taktvorgabe möglich, deren Änderung in diesem Fall das Scheduling nicht beeinträchtigten. Die Ergebnisse aus Tabelle A.13 zeigen bei Unterscheidung der Übersetzungsoptionen in Bild 6.41 eine Vergrößerung der CLB-Anzahl für die `incremental_map` und neutrale Wirkung für die `min_paths`-Option. Die in Bild 6.42 hervorgehobene Flächenvorgabe läßt auch bei diesem Entwurf keinen zielgerichteten Einfluß erkennen. Die Strukturoptimierungen, die in Bild 6.43 markiert sind, zeigen keine Wirkung auf die Ergebnisse, die mit oder ohne Optionen gleich ausfallen und sich daher in der Graphik überlagern.

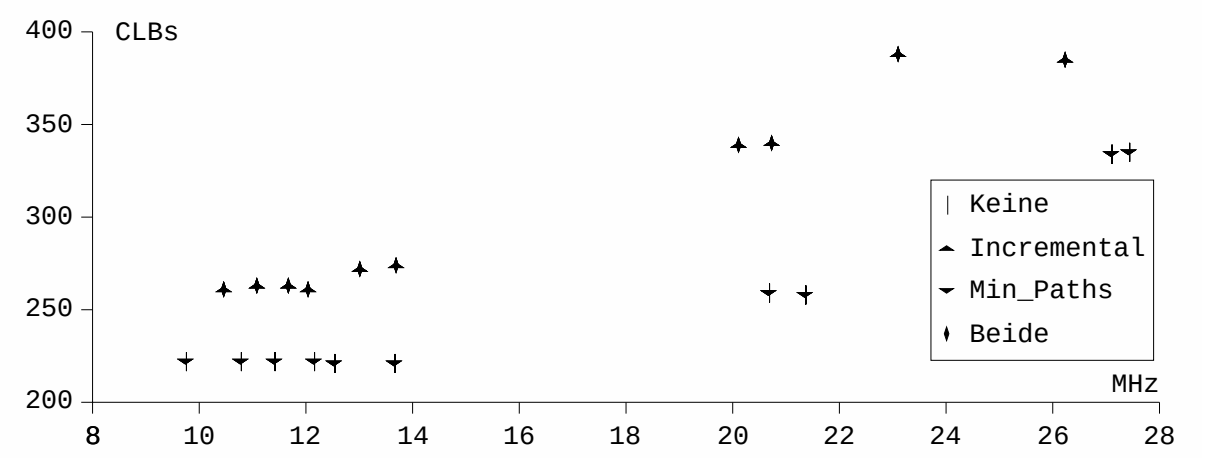


Bild 6.41: `compile`-Optionen nach der High-Level-Synthese des MRISC

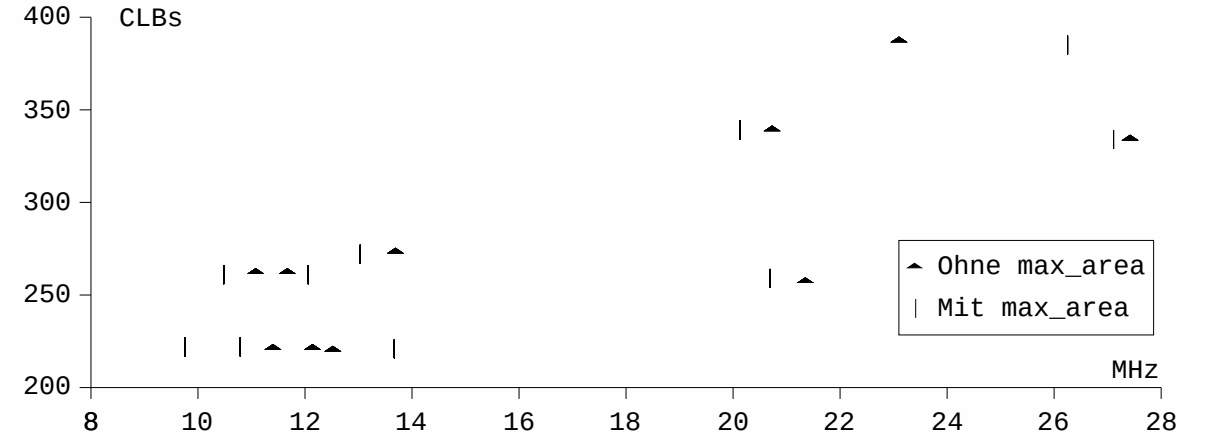


Bild 6.42: `set_max_area` nach der High-Level-Synthese des MRISC

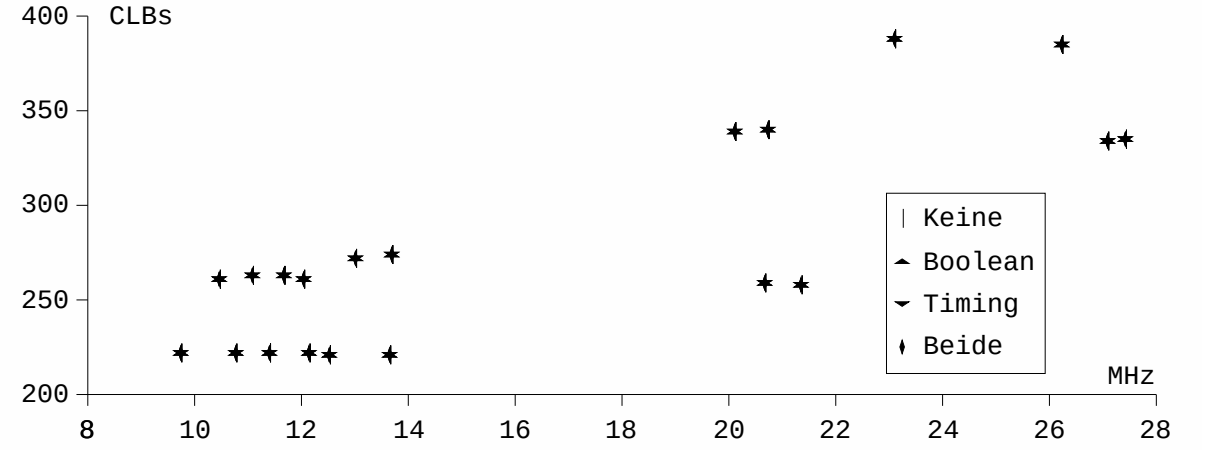


Bild 6.43: Strukturoptimierungen nach der High-Level-Synthese des MRISC

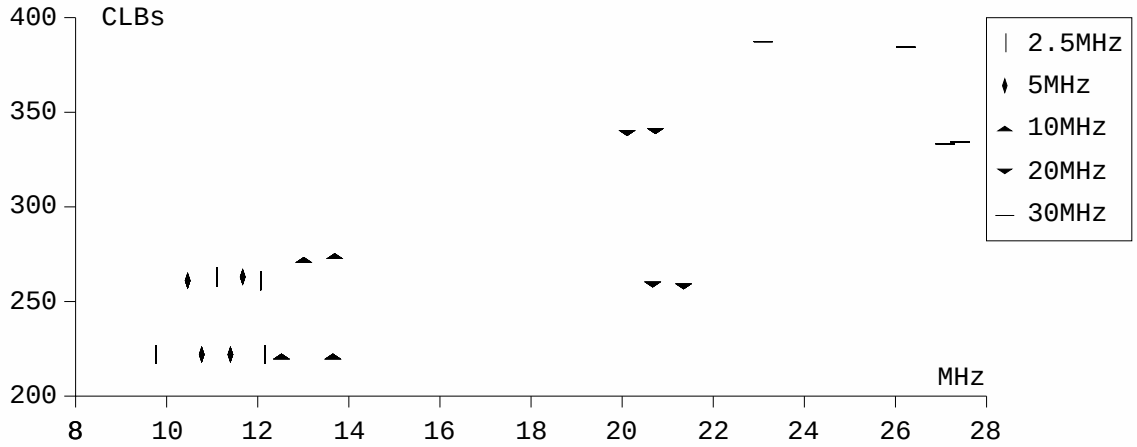


Bild 6.44: Taktvorgaben bei der High-Level-Synthese des MRISC

Für die Taktvorgaben deutet sich in Bild 6.44 eine direkte Abhängigkeit in Form zweier Kennlinien an, die sich in den `compile`-Optionen unterscheiden. Um dies zu verifizieren und näher zu analysieren wurden in einer weiteren Untersuchung die Taktvorgaben von 0,2MHz bis 40MHz in Schritten von 200kHz variiert, wobei eine minimale Flächenvorgabe und beide Strukturoptimierungen aktiviert waren. Während die Übersetzungsoption `min_paths` deaktiviert war, wurde die Option `incremental_map` hierbei in an- und abgeschaltetem Zustand betrachtet, so daß die Ergebnisse in Tabelle A.14 entstanden.

Für die Geschwindigkeit (Bild 6.45) ergibt sich für die Vorgaben bis 11MHz ein Plateau ähnlich schneller Ergebnisse um 11MHz, an das sich im Vorgabebereich von 11MHz bis 25MHz ein nahezu lineares Wachstum bis hin zu erreichten 25MHz anschließt. Bei Vorgabenwerten von 25MHz bis 30MHz findet der Scheduler keine Lösung in den möglichen Operatorkombinationen und liefert keine Ergebnisse. Ab Vorgaben von 30MHz entstehen ungeordnet Ergebnisse, die Taktraten von 25MHz bis 32MHz erlauben. Die Option `incremental_map` wirkt sich nur wenig aus, die Ergebnisbereiche der Optionsvarianten überlagern sich weitgehend.

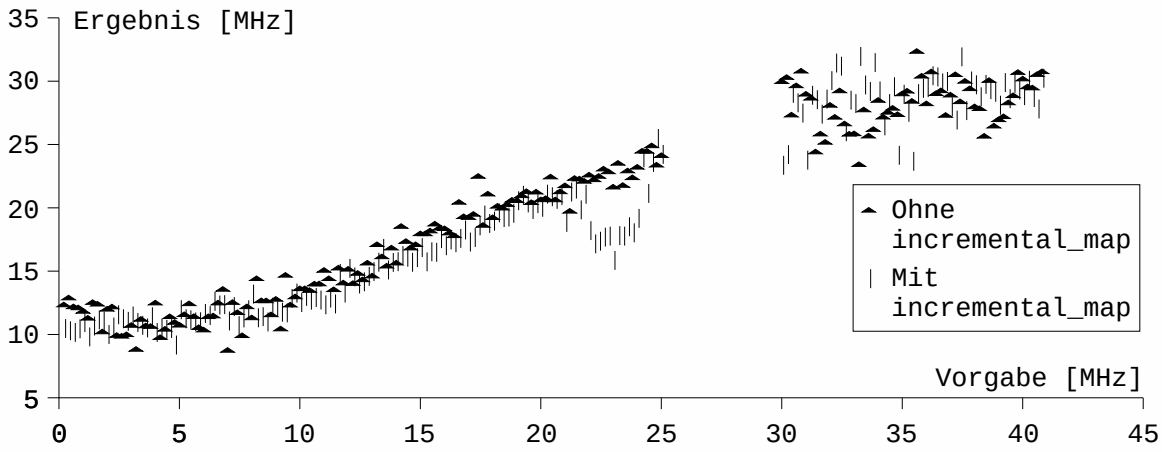


Bild 6.45: Einfluß von Taktvorgaben auf die Geschwindigkeit bei HL-MRISC

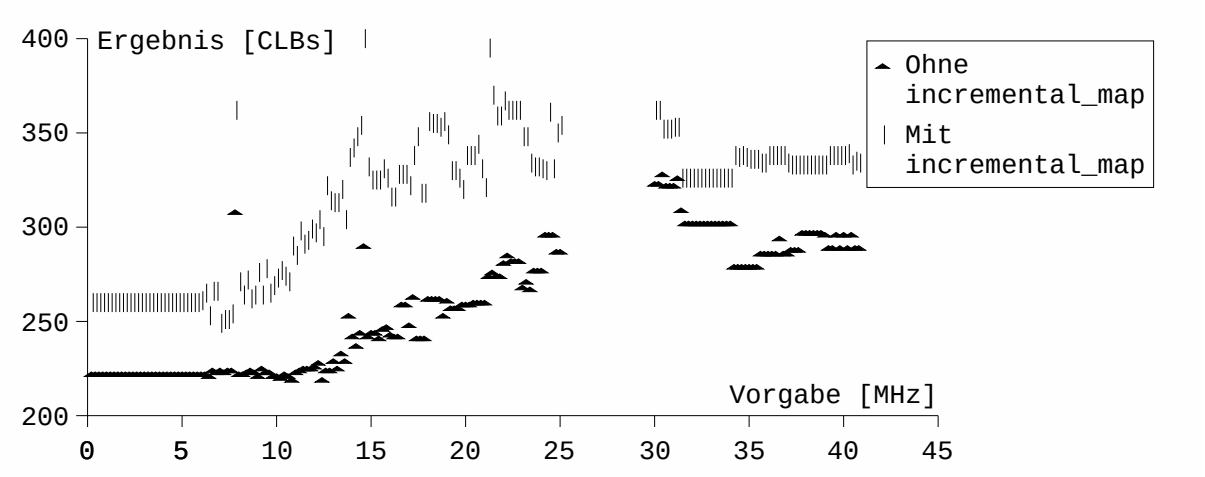


Bild 6.46: Einfluß von Taktvorgaben auf die Schaltungsgröße bei HL-MRISC

Die Schaltungsgröße in Bild 6.46 ist für Vorgaben bis 6MHz konstant und wächst danach an, wobei sich für einzelne Ergebnisse erhebliche Sprünge in positiver oder negativer Richtung ergeben. Ab Forderungen von 30Mhz sinkt die Größe zunächst und bleibt danach mit kleineren Sprüngen weitgehend stabil. Durch die Option `incremental_map` entstehen größere Schaltungen als ohne sie und damit zwei deutlich getrennte Kennlinien.

Zusammenfassung der MRISC-Synthese: Die Darstellung aller Ergebnisse der RTL- und High-Level-Synthese in Bild 6.47 zeigt, daß für Taktraten bis 25MHz die RTL-Synthese etwas kleinere Ergebnisse liefert, ihr Ergebnisbereich sich aber mit dem der High-Level-Synthese überschneidet. Für Taktraten über 25Mhz kann nur die High-Level-Synthese durch die Verteilung zeitkritischer kombinatorischer Operationen auf mehrere Takte Ergebnisse liefern. Die RTL-Synthese könnte dies nur bei expliziter manueller Spezifikation, sofern der modellierte Datenfluß eine solche Verteilung überhaupt zuläßt.

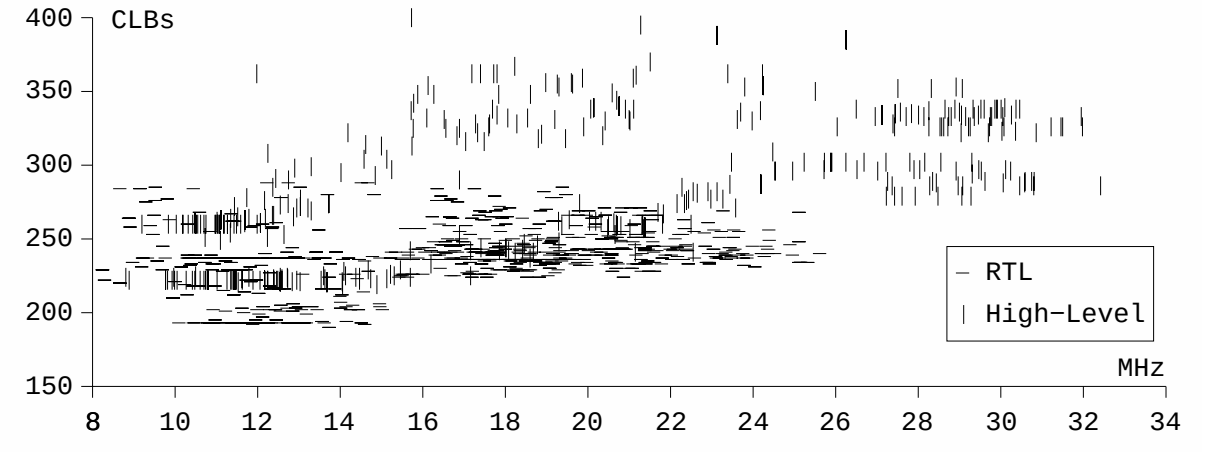


Bild 6.47: Ergebnisse der MRISC-Synthesemethoden

Die High-Level-Synthese erfüllt die Anforderungen der Aufgabe insgesamt besser als die RTL-Synthese, da sie für das Ziel-FPGA die schnellsten Ergebnisse liefert

und eine geringere Entwurfzeit erforderte. Der Grund hierfür liegt in den beim MRISC vorhandenen arithmetischen Operationen und Registern, die durch die MRISC-Spezifikation nicht auf einzelne Takte festgelegt sind. Das Scheduling der High-Level-Synthese kann unter Berücksichtigung des benötigten Datenflusses Operationen und Registerzugriffe beliebig im Taktraster verteilen und auch über mehrere Takte ablaufen lassen.

Während bei der RTL-Synthese die Forderung zu hoher Taktraten schlechtere Ergebnisse liefert als die Vorgabe eines nah am erreichbaren Maximum liegenden Wertes, wirkt sich dies bei der High-Level-Synthese nicht aus. Lediglich die Lücke im Ergebnisbereich ist störend, da sie bei iterativer Bestimmung der optimalen Vorgabe als Obergrenze der erreichbaren Werte fehlinterpretiert werden kann. Für beide Synthesemethoden sind deutliche Sprünge in den Ergebniswerten bei kleinen Änderungen der Vorgaben vorhanden, so daß eine iterative Bestimmung der maximal erreichbaren Taktrate mit leichter Variation der Vorgabe in beiden Fällen für optimale Ergebnisse notwendig ist.

Die bereits bei den vorangegangen Entwurfsuntersuchungen des discount und des DAR beobachteten generellen Eigenschaften und Auswirkungen der Syntheseoptionen bestätigten sich auch bei den MRISC-Ergebnissen.

6.2.4 Entwurf eines Chipkartenlesers

Für einen Controller zur Auswertung der Chipkarten öffentlicher Telefone gab es nach der Konstruktion in einem Entwurfspraktikum mit Synthese [Blinze99A] neben der Musterlösung mit Controllersynthese zahlreiche Varianten in RTL-Verilog. Dies ermöglichte den Vergleich zwischen sehr unterschiedlichen Lösungen zu einer Entwurfsaufgabe.

Aus den Praktikumslösungen wurde ein Entwurf stellvertretend ausgewählt, welcher der Musterlösung in den Ausgabemeldungen für die Chipkartendaten so genau entspricht, daß für einen Anwender kein Unterschied erkennbar ist. Sowohl an die Musterlösung mit Controllersynthese als auch an die Praktikumslösung mit RTL-Synthese wurden für die Syntheseexperimente diese Anforderungen gestellt:

- Xilinx-FPGA 4013E-4 als Zielhardware
- Taktung mit 1MHz
- I/O-Protokolle für Chipkarte, LC-Display und Zeichensatz-ROM nur im Signalverlauf spezifiziert, keine vorgeschriebene Bindung an Takte
- Einmaliger Durchlauf durch die Kontrollsequenz, dabei mehrere mehrfach benutzte Teilsequenzen

Der Entwurf ist trotz der Verarbeitung vieler Daten angebundener Komponenten kontrollflußdominiert und benutzt als komplexesten Operator einen fünfstelligen BCD-Addierer zur Berechnung des Kartenwertes.

RTL-Synthese des Chipkartenlesers: Der aus mehreren Teilmodulen bestehende RTL-Entwurf, dessen Entwurfszeit bei etwa 9 Mannwochen lag (3 Praktikanten in drei Wochen), wurde global synthetisiert und dabei in der Flächenvorgabe, den Strukturoptimierungen, den `compile`-Optionen und der Taktvorgabe variiert.

Die schnellsten Schaltungen der in Tabelle A.16 aufgelisteten Ergebnisse der RTL-Synthese entstehen nach Bild 6.48 mit Verwendung einer Flächenvorgabe. Da aber alle Ergebnisse dieser Syntheseversuche die Forderung von 1MHz bei weitem übertreffen, die schnellsten Schaltungen zu den größten gehören und eine Flächenvorgabe auch zu langsamen, kleinen Schaltungen führen kann, ist diese Option hier nicht nutzbringend und zielgerichtet anwendbar.

Die Unterscheidung der Ergebnisse nach Strukturoptimierungen in Bild 6.49 zeigt, daß mit diesen Optimierungen die kleinsten Schaltungen entstehen, diese aber auch zu den langsameren gehören. Während die Geschwindigkeit unkritisch ist, kann mit diesen Optimierungen der Logikverbrauch unter 400 CLBs gebracht werden, womit die Schaltung auf das nächst kleinere und billigere FPGA passen würde. Mit der Booleschen Optimierung ergeben sich etwas kleinere Schaltungen als nur mit der Timing-Optimierung und insgesamt auch die kleinsten Ergebnisse.

Aus den `compile`-Optionen (Bild 6.50) ist keine grundlegende Wirkung auf die Größe oder Geschwindigkeit ablesbar. Die Option `min_paths` ist wirkungslos, da sich ihre Ergebnisse mit denen ohne Optionen erzielten vollständig überlagern.

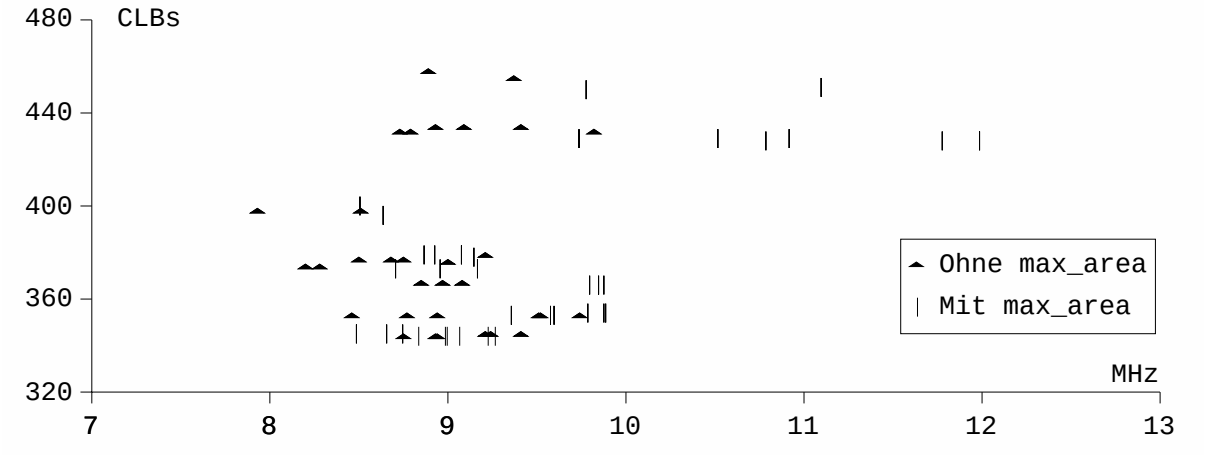


Bild 6.48: `set_max_area` bei der RTL-Synthese des Chipkartenlesers

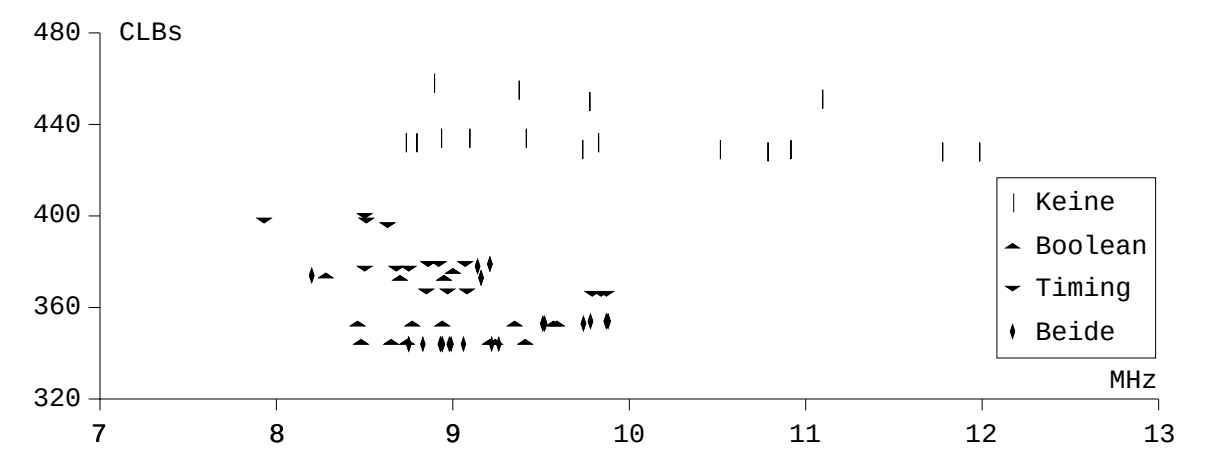


Bild 6.49: Strukturoptimierungen bei der RTL-Synthese des Chipkartenlesers

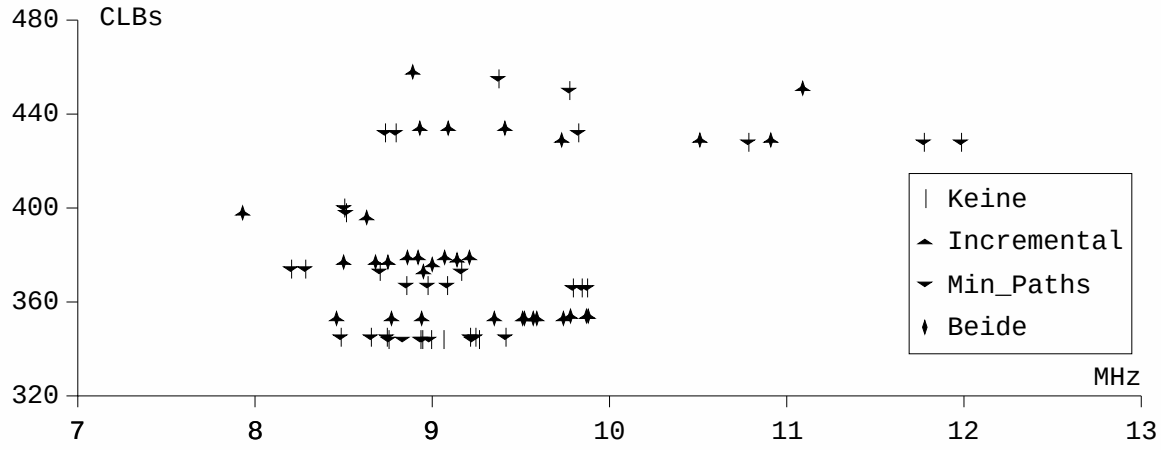


Bild 6.50: `compile`-Optionen bei der RTL-Synthese des Chipkartenlesers

Die Taktvorgaben in Bild 6.51 führen unabhängig vom Vorgabewert zu kleineren Schaltungen als Syntheseläufe ohne Taktspezifikation. Die Angabe eines Taktes bewirkt offenbar eine intensivere Flächenoptimierung als kein Takt, wenn eine Zeitoptimierung nicht mehr notwendig ist.

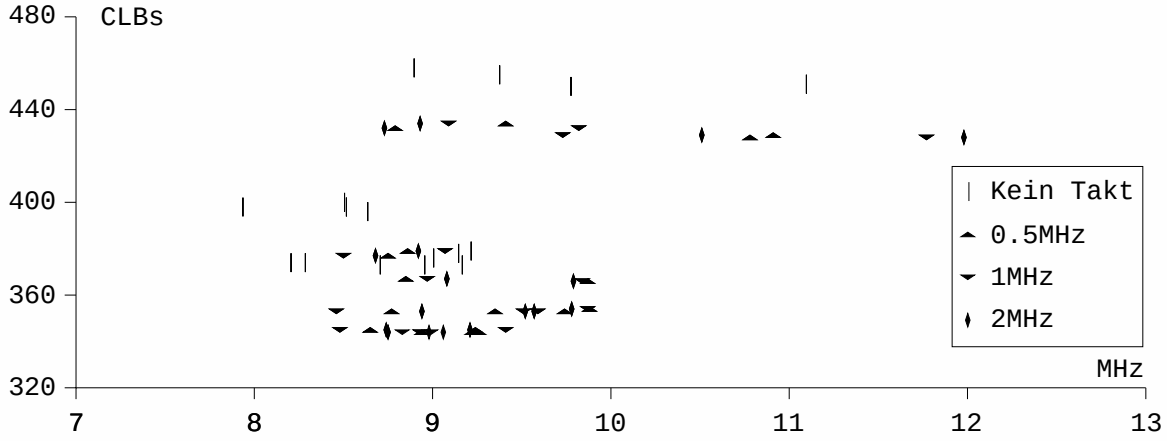


Bild 6.51: Taktvorgaben bei der RTL-Synthese des Chipkartenlesers

Der Zusammenhang zwischen der Taktvorgabe und der Geschwindigkeit bzw. der Größe der resultierenden Schaltung wurde im Vorgabebereich von 0,5 bis 16,5 MHz in Schritten zu 500kHz genauer untersucht. Hierbei waren eine minimale Flächenvorgabe, beide Strukturoptimierungen und keine Übersetzungsoptionen aktiviert, so daß sich die Ergebnisse in Tabelle A.17 ergaben. In den erreichten Taktraten (Bild 6.52) werden für keine Vorgabe (0MHz) und Vorgaben unterhalb von 9MHz Werte um ein Plateau von 9MHz herum erreicht. Vorgaben zwischen 9 und 13MHz führen zu einer linear folgenden Entwicklung der Ergebnistaktes. Darüber hinaus gehende Vorgaben bleiben bis 15MHz auf einem Ergebnisplateau von 13MHz und führen danach zu absinkenden Taktraten durch Überforderung der Optimierungen.

Bei der Schaltungsgröße (Bild 6.53) werden für keine Vorgaben (0MHz) die größten Schaltungen erzeugt. Bei Taktvorgaben unterhalb von 6MHz entstehen die kleinsten Schaltungen, darüber hinaus erfolgt zunächst ein Wachstum bis zu einem Plateauwert von etwa 354 CLBs für Vorgaben bis 12MHz, der auch bei größeren Vorgabewerten nicht übertroffen wird.

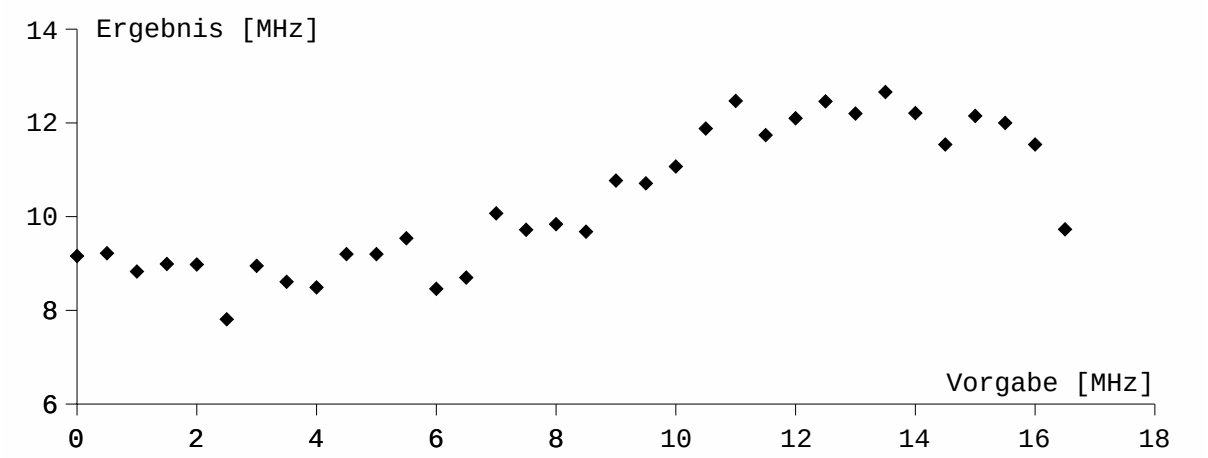


Bild 6.52: Einfluß von Taktvorgaben auf den kritischen Pfad des Chipkartenlesers

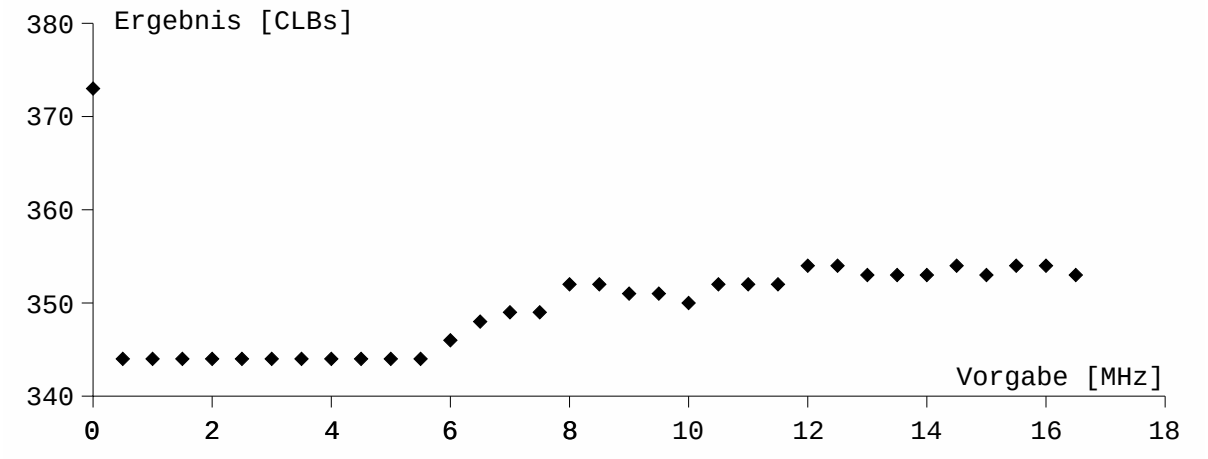


Bild 6.53: Einfluß von Taktvorgaben auf die Größe des Chipkartenlesers

Controllersynthese des Chipkartenlesers: In der Protocol-Compiler-HDL setzt sich der Chipkartenleser aus mehreren parallel arbeitenden, kommunizierenden Teilcontrollern auf der obersten Entwurfsebene zusammen. Neben der Entwurfshierarchie besitzen diese Teilcontroller eine Kommunikationshierarchie, welche zur Mehrfachnutzung einiger Abläufe und Ressourcen eingesetzt wird. Hierdurch konnte der Entwurf einerseits klein gehalten werden und andererseits in etwa drei Wochen durchgeführt werden.

Die Zustandskomplexität des Chipkartenlesers verhinderte eine Betrachtung aller bei der Controllersynthese möglichen Codierungsvarianten, da die meisten Codierungen eine vollständige Zustandsanalyse voraussetzen. Diese erwies sich als undurchführbar. Es waren lediglich die verteilte Codierung (Distributed), die

manuelle Partitionierung (Partitioned) sowie die automatische Codierungswahl (Automatic) möglich, deren Ergebnisse in Bild 6.54 entsprechend der Werte aus Tabelle A.18 unterschieden werden.

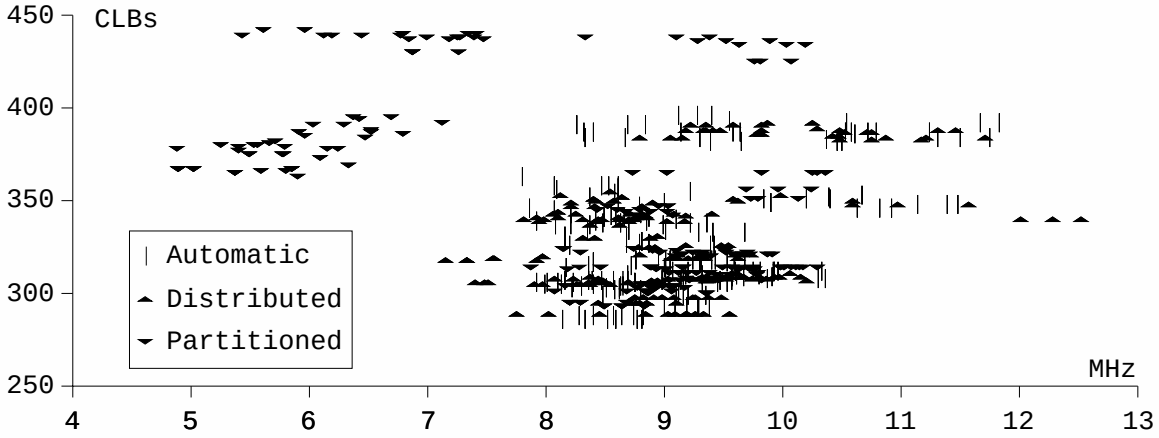


Bild 6.54: Codierungsvarianten der Controllersynthese des Chipkartenlesers

Bei der automatischen Codierungswahl wurde aufgrund der Analysekomplexität eine verteilte Codierung ohne Unterpartitionen ausgewählt, die mit der direkt gewählten Distributed-Codierung allerdings nicht identisch ist. Die Ergebnisse der Automatic- und der Distributed-Codierung überschneiden sich weitgehend und sind die kleinsten und die schnellsten Entwürfe. Die manuelle Partitionierung liefert im Vergleich zwar die langsamsten und größten Schaltungen, ist aber auch mit vielen Ergebnissen im Mittelfeld der Geschwindigkeit und bei den sehr kleinen Schaltungen vertreten. Ihr Hauptvorteil liegt in der Verkleinerung des Zustandsraumes für die übrigen Partitionscodierungen und damit reduzierten Synthesezeiten. Beim Chipkartenleser wurden aus diesem Grund One-Hot-Codes für die vergleichsweise kleinen Interface-Controller des LCD-Busprotokolls und der Kartenwertsummierung gewählt, die keine Zustandsanalyse erfordern.

Bei den Modellstrukturen (Bild 6.55) liefert die Single-Process-Architektur in der Tendenz kleinere Ergebnisse als die Split-Process- oder Multi-Process-Arten und insgesamt auch die kleinsten Schaltungen.

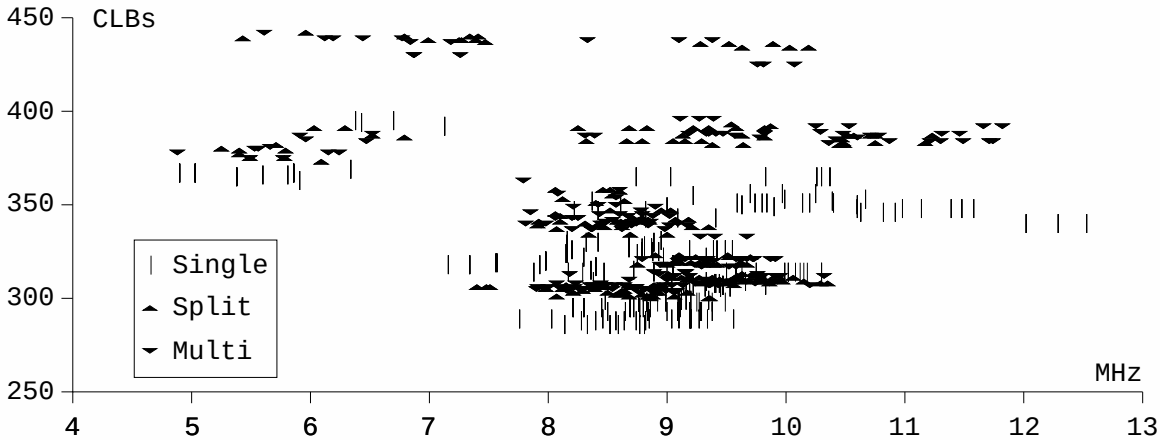


Bild 6.55: Modellstrukturen der Controllersynthese des Chipkartenlesers

Die Verwendung einer Flächenvorgabe hat in Bild 6.56 keine erkennbare Wirkung auf die Schaltungsgröße, mit ihr entstehen aber die schnellsten Schaltungen. Durch die von allen Ergebnissen bei weitem übertroffene, geforderte Taktrate ist dieser Effekt allerdings nicht nutzbringend anwendbar.

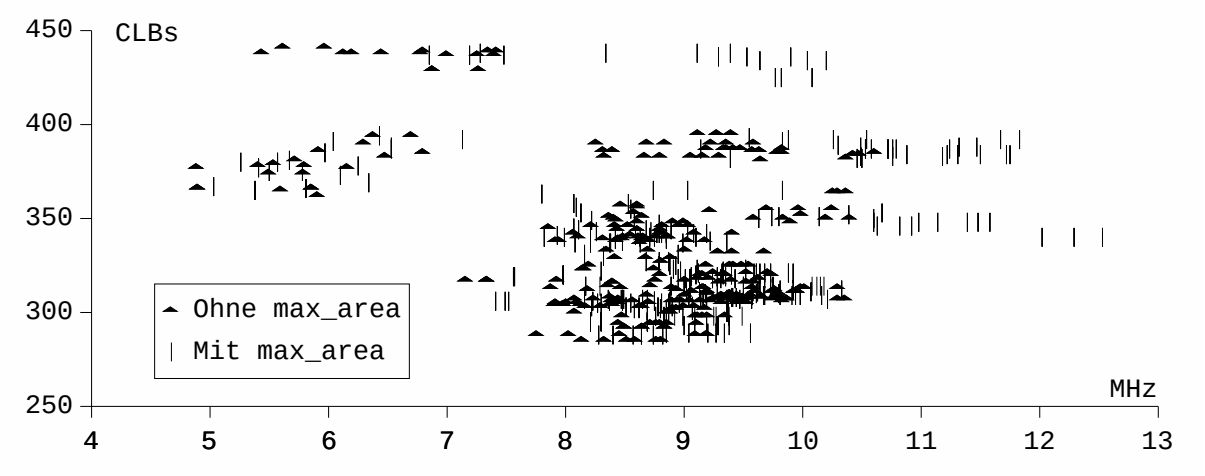


Bild 6.56: `set_max_area` nach der Controllersynthese des Chipkartenlesers

Die in Bild 6.57 betrachteten Strukturoptimierungen zeigen für beide Optionen eine deutliche Verringerung der Größe, wobei sich die Boolesche Optimierung stärker auswirkt. Die Schaltungsgeschwindigkeit sinkt mit diesen Optimierungen, was aufgrund der zeitunkritischen Taktrate für die Schaltung ohne praktische Bedeutung ist.

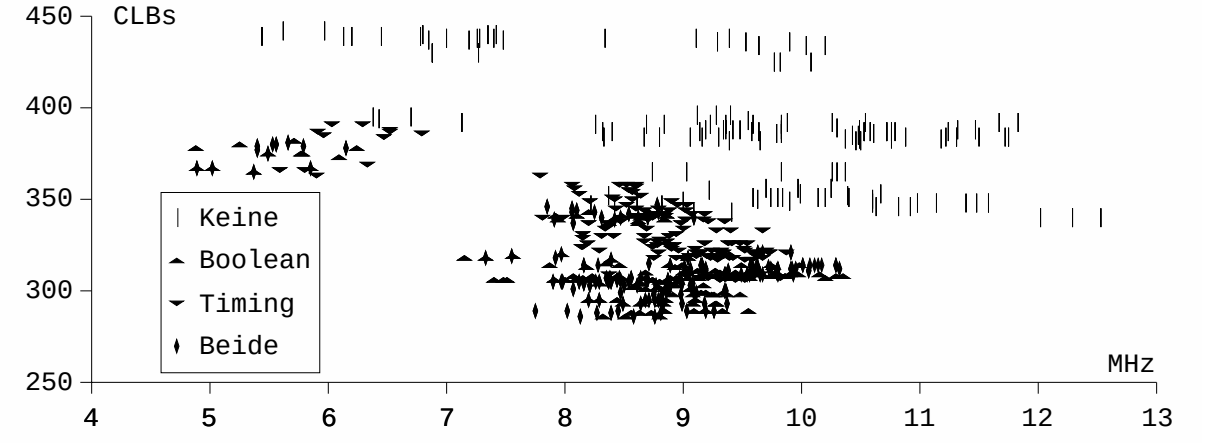


Bild 6.57: Strukturoptimierungen nach der Chipkartenleser-Controllersynthese

Aus den Übersetzungsoptionen, die in Bild 6.58 gekennzeichnet sind, ergibt sich keine eindeutige Tendenz für die Größe oder die Geschwindigkeit der Ergebnisse. Die jeweils vollständige Überlagerung der Werte für keine Option und `min_paths` bzw. beide Optionen und `incremental_map` deutet auf die Wirkungslosigkeit der `min_paths`-Option hin.

Die Unterscheidung der Ergebnisse nach Taktvorgaben (Bild 6.59) zeigt, daß die kleinsten und die schnellsten Ergebnisse nur bei der Angabe eines Taktes erreicht werden. Die langsamsten und die größten Ergebnisse entstehen ohne eine

Taktvorgabe. Durch die Angabe eines Taktes erfolgt eine bessere Optimierung der Schaltungsgröße als ohne Taktvorgabe. Eine Zeitoptimierung ist hierbei nicht erforderlich, da die geforderte Mindesttaktrate optimierungsfrei übertroffen wird.

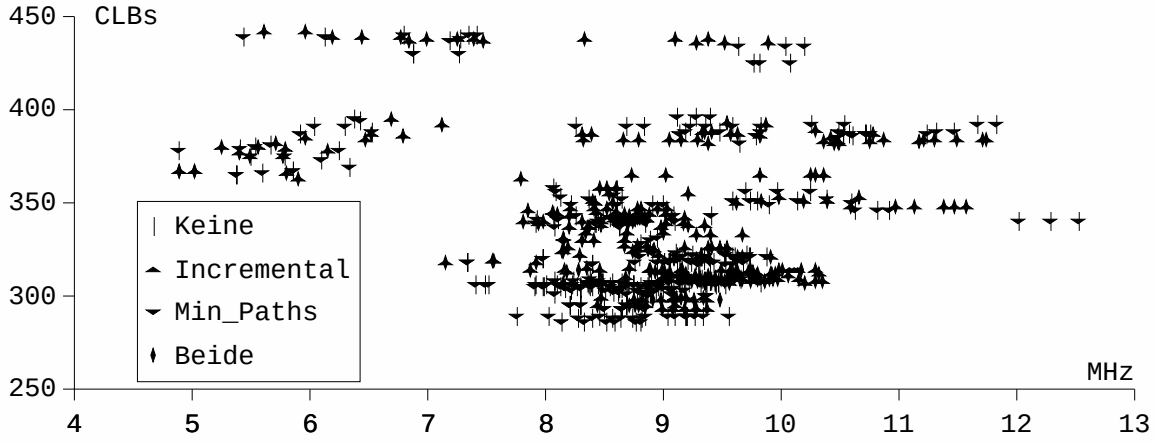


Bild 6.58: `compile`-Optionen nach der Controllersynthese des Chipkartenlesers

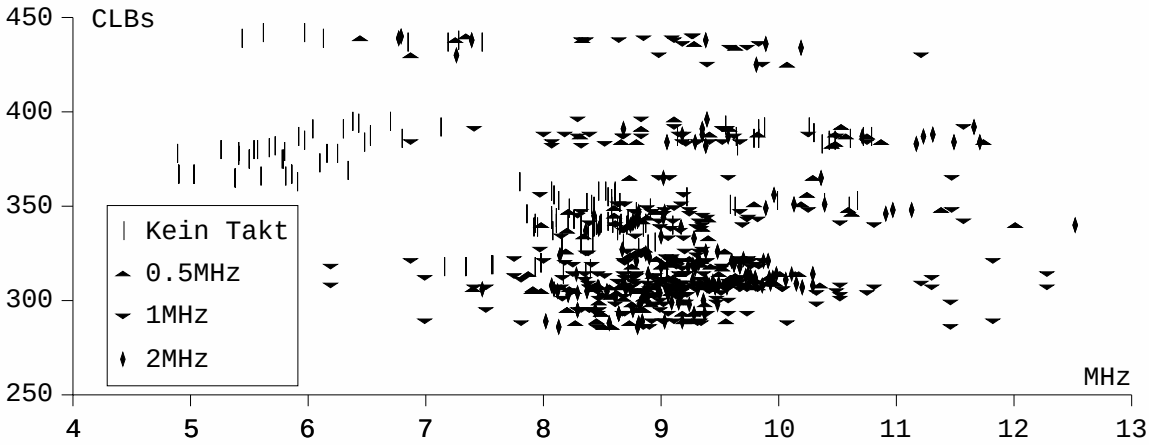


Bild 6.59: Taktvorgaben nach der Controllersynthese des Chipkartenlesers

Zusammenfassung der Chipkartenleser-Synthese: In der Gegenüberstellung der Ergebnisse für den Chipkartenleser (Bild 6.60) ist erkennbar, daß die Controllersynthese nicht nur einen größeren Ergebnisraum besitzt als die RTL-Synthese, sondern auch kleinere und ähnlich schnelle Schaltungen produziert. Aufgrund der deutlich geringeren Entwurfszeit mit der Controllersynthese ist diese für den Chipkartenleser die bessere Entwurfsmethode.

Die Kombination aus erreichbarer Schaltungsgröße und Geschwindigkeit der Controllersynthese eröffnet außerdem die Möglichkeit zur Verwendung kleinerer, langsamerer und damit kostengünstigerer FPGAs.

Die Beobachtungen aus den Syntheseversuchen des discount, des DAR und des MRISC für die Syntheseoptionen bestätigen sich im wesentlichen auch beim Chipkartenleser. Für die Flächenoptimierung ist die Wirkung hin zu schnelleren Schaltungen bei unkritischem Timing auffällig, jedoch nicht nutzbar. Bei den

Taktvorgaben ist die verbesserte Flächenoptimierung bemerkenswert, die anstelle der nicht benötigten Zeitoptimierung stattfindet. Die Angabe eines Taktes kann die Ergebnisqualität also auch dann verbessern, wenn keine Zeitoptimierung erforderlich ist und ist somit auch für nicht zeitkritische Entwürfe zu empfehlen.

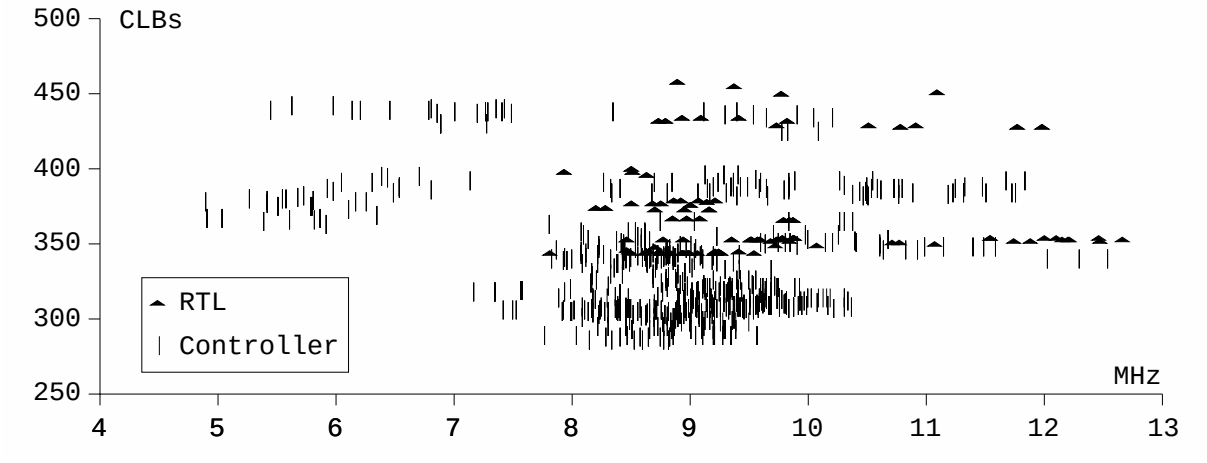


Bild 6.60: Ergebnisse der Chipkartenleser-Synthesemethoden

6.2.5 Entwurf eines Kryptographie-Datenpfades

Die bei den bereits vorgestellten Entwürfen beobachteten Möglichkeiten von RTL- und High-Level-Synthese sowie die dabei entstandenen Abwägungen wurden an einem Kryptographie-Datenpfad für das IDEA-Verfahren [Ascom98] verifiziert. In dem Kryptographie-Datenpfad werden acht gleich aufgebaute Verschlüsselungsstufen regulär aneinandergereiht, wobei in einer Stufe mehrere Multiplikationen, Additionen und XOR-Verknüpfungen auf 16-Bit Daten stattfinden (Bild 6.61).

Die vergleichsweise große Anzahl an arithmetischen Operationen, die datenflußdominierte Verarbeitung und die nicht fest vorgegebene Verarbeitungszeit ließen auf der Grundlage der vorangegangenen Experimente besonders gute Ergebnisse der High-Level-Synthese gegenüber der ebenfalls untersuchten RTL-Synthese erwarten.

Die Komplexität des vollständigen IDEA-Verschlüsselungspfades hätte in der zur Verfügung stehenden Zeit jedoch nur die Durchführung weniger Variationsmöglichkeiten erlaubt. Zugunsten eines breiteren Spielraumes der Syntheseeinstellungen wurde daher die Regularität des Datenpfades ausgenutzt und die Synthese auf eine der acht gleichartigen Stufen beschränkt.

Sowohl bei der RTL- als auch bei der High-Level-Synthese wurden unterschiedliche Einstellungen für die Flächenvorgabe, die Strukturoptimierungen, die Übersetzungsoptionen und die Taktvorgabe gewählt. An die RTL- und die High-Level-Beschreibung wurden zudem die gleichen Anforderungen gestellt:

- Xilinx-FPGA 4052XL-3 als Zielhardware
- möglichst kurze Verarbeitungsdauer der Daten
- Anliegen der Eingangsdaten über die gesamte Verarbeitungsdauer

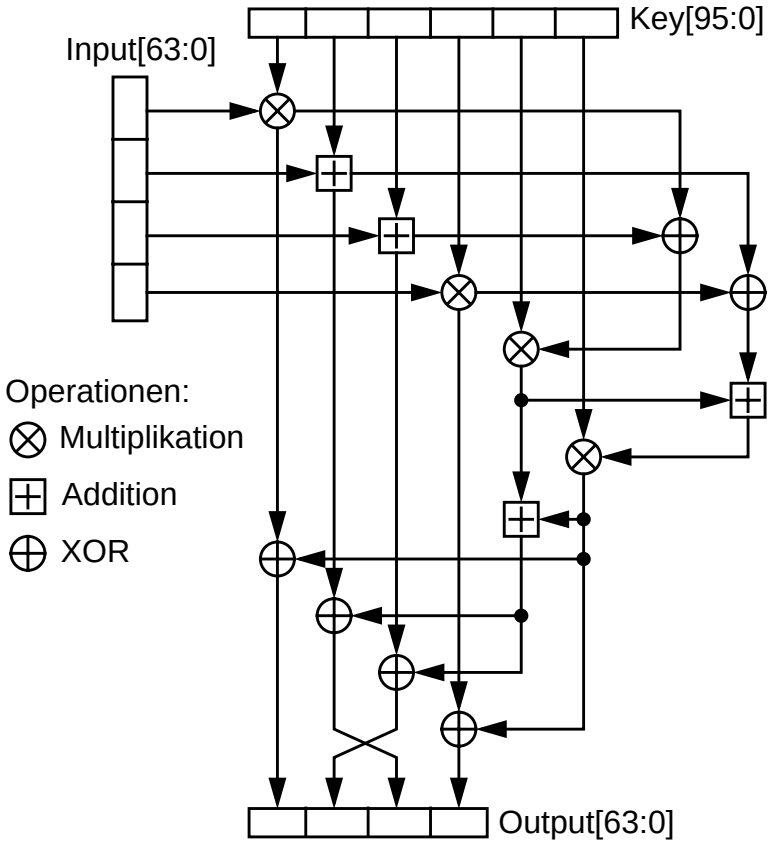


Bild 6.61: Datenfluß in einer IDEA-Stufe

Die Verilog-Modelle für RTL- und High-Level-Synthese sind in weiten Teilen ähnlich gestaltet. Während im RTL-Modell die Multiplikation Modulo 2^16+1 ein Untermodul ist, dessen Instanzen in Continuous-Assignments eingebettet sind, wird im High-Level-Modell dafür eine Unterfunktion im Ablauf prozeduraler Zuweisungen aufgerufen. Die Entwurfszeit betrug aufgrund der Ähnlichkeit und der Ableitbarkeit voneinander für beide Modelle zusammen etwa eine halbe Woche, wobei keines der Modelle Vorteile in der Entwurfsdauer besaß.

RTL-Synthese einer IDEA-Stufe: Der RTL-Entwurf besteht aus einem Hauptmodul und einem Multiplikationsmodul. Die Multiplikation wird im Hauptmodul eingebettet in Continuous-Assignments mit Additionen und XOR-Verknüpfungen entsprechend der Struktur in Bild 6.61 instanziert. Die Ausgangsdaten werden in Registern gespeichert, die bei positiven Taktflanken neu geladen werden, so daß bei regulärer Anreihung mehrerer Stufen eine Pipeline entsteht. Die Ergebniswerte der RTL-Synthese sind in Tabelle A.20 dokumentiert.

Eine bezüglich der Flächenvorgabe differenzierte Darstellung der Ergebnisse in Bild 6.62 zeigt ebenso wie die Unterscheidung nach Strukturoptimierungen (Bild 6.63) keine gezielten Auswirkungen auf die Größe oder die Geschwindigkeit, was bei den Strukturoptimierungen der RTL-Synthese bisher anders war.

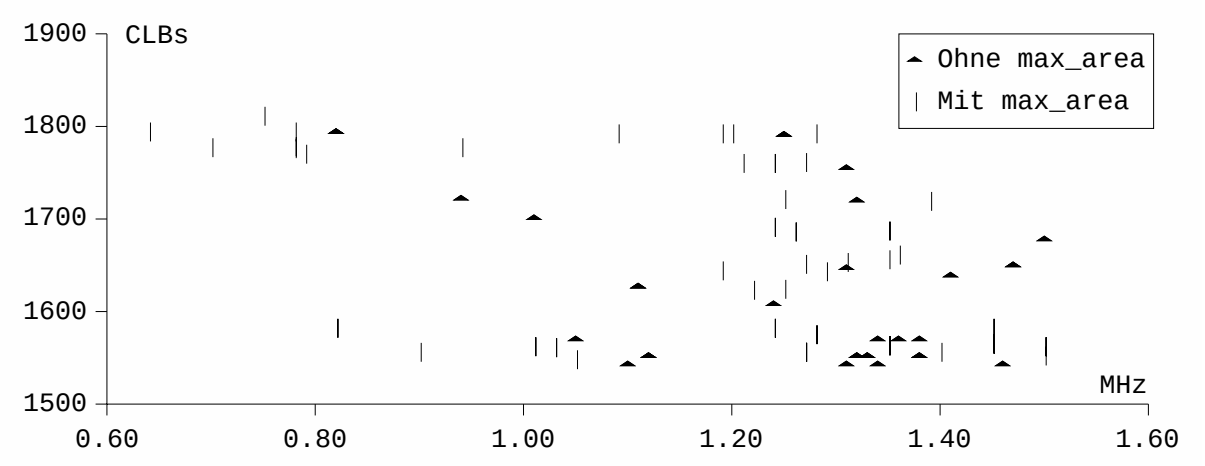


Bild 6.62: `set_max_area` bei der RTL-Synthese der IDEA-Stufe

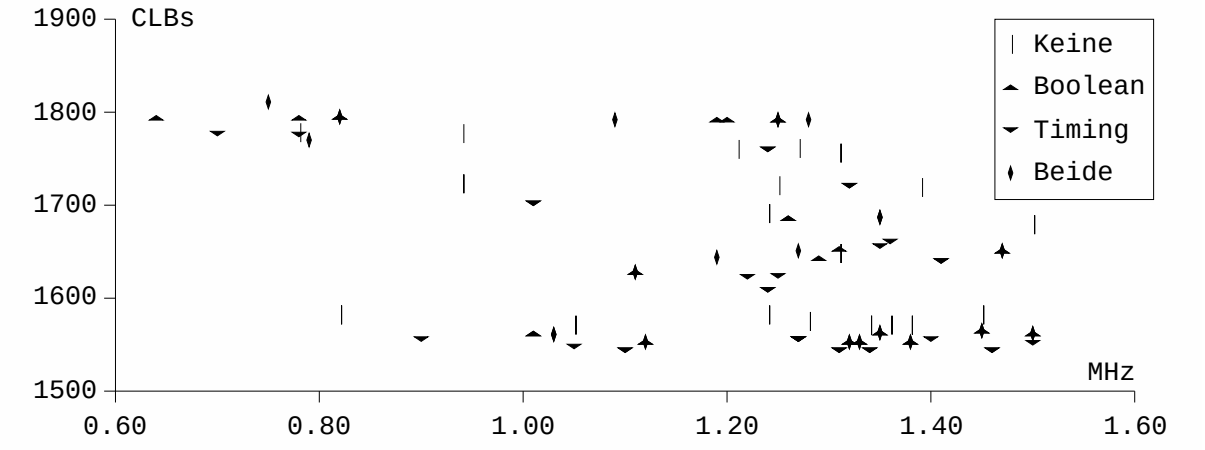


Bild 6.63: Strukturoptimierungen bei der RTL-Synthese der IDEA-Stufe

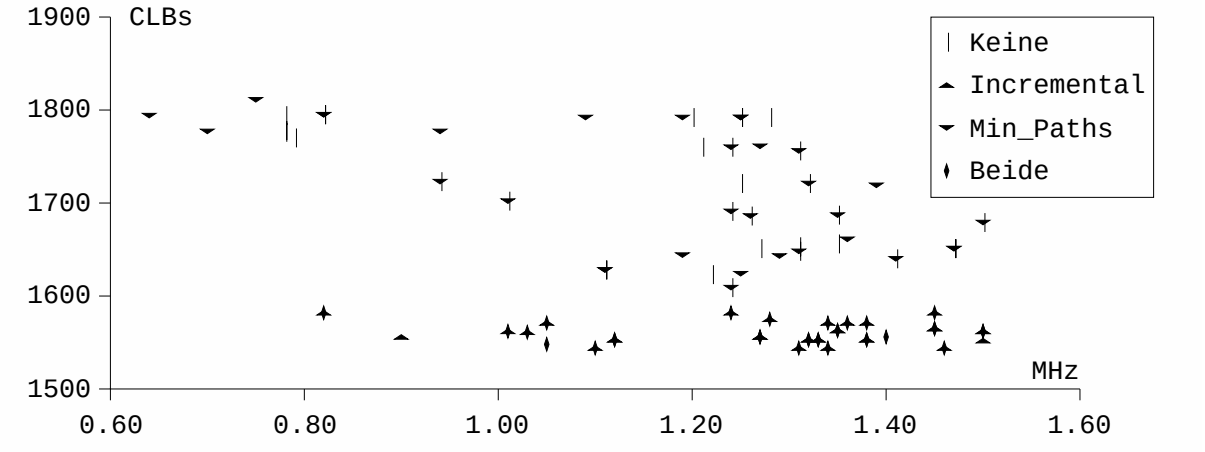


Bild 6.64: `compile`-Optionen bei der RTL-Synthese der IDEA-Stufe

Bei den Übersetzungsoptionen ist im Gegensatz zu den Entwürfen der vorigen Abschnitte ebenfalls eine veränderte Wirkung festzustellen, da die Ergebnisse mit der `incremental_map`-Option hier grundsätzlich kleiner sind als ohne diese Option (Bild 6.64). Bei den Entwürfen der vorigen Abschnitte war bei der direkten RTL-Synthese keine zielgerichtete Wirkung der Übersetzungsoptionen auf Geschwindigkeit oder Größe erkennbar.

Der Grund hierfür ist in der Modulstruktur und im Übersetzungsablauf zu vermuten. Die Multiplikation wird mehrfach instanziert und deswegen zunächst einzeln übersetzt, danach vervielfältigt und im Kontext mit der gesamten IDEA-Stufe erneut übersetzt, wodurch jede Multiplikationsinstanz zwei Übersetzungen durchläuft. Die inkrementelle Optimierung besitzt damit einen Übersetzungslauf als Grundlage.

In Bild 6.65 wird durch die Markierung der Taktvorgaben eine Aufteilung des Ergebnisfeldes in Teilbereiche erkennbar, wobei 1MHz die schnellsten Ergebnisse liefert und 2MHz die langsamsten und größten. Eine genauere Untersuchung des Einflusses der Taktvorgabe von 0,1 bis 2,3 MHz in Abstufungen zu 100kHz, deren Ergebnisse in Tabelle A.21 zusammengefaßt sind, zeigt für die Geschwindigkeit (Bild 6.66) und die Schaltungsgröße (Bild 6.67) deutliche Abhängigkeiten.

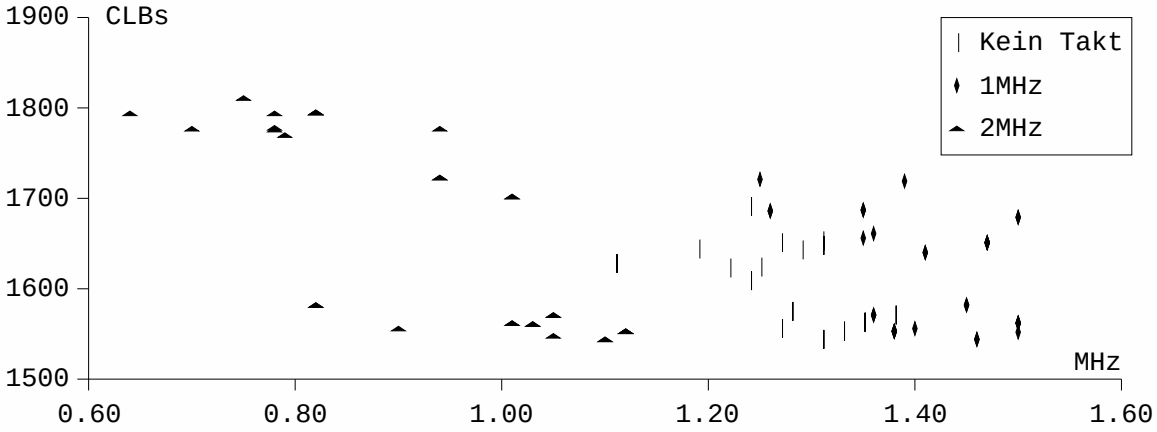


Bild 6.65: Taktvorgaben bei der RTL-Synthese der IDEA-Stufe

Die erreichte Taktrate in Bild 6.66 schwankt bei Vorgaben von 100 bis 800kHz um ein Plateau von 1,4MHz, während ohne Taktvorgabe (dargestellt bei 0MHz) nur etwa 1,2MHz erreicht werden. Mit wachsenden Vorgaben über 800kHz sinkt die erreichte Taktrate dann durch Überforderung der Optimierung bis zu 700kHz ab. Der Unterschied zwischen der maximal erreichten Taktrate und dem in diesem Fall vorgegebenen Taktwert ist auf zu ungenaue Abschätzungen der realen FPGA-Laufzeiten für Logik und Verdrahtung während der Optimierung zurückzuführen.

Für die Schaltungsgröße wird ohne Taktvorgabe das kleinste Ergebnis erzielt und zwischen 0,1 und 1MHz fast konstante Werte. Mit wachsenden Taktvorgaben über 1MHz steigt die Schaltungsgröße an, bis sie sich schließlich bei etwa 1800 CLBs einpendelt. Im Gegensatz zu anderen Entwürfen führt die Taktvorgabe bei der IDEA-Stufe also nicht zu einer Verbesserung der Flächenoptimierung.

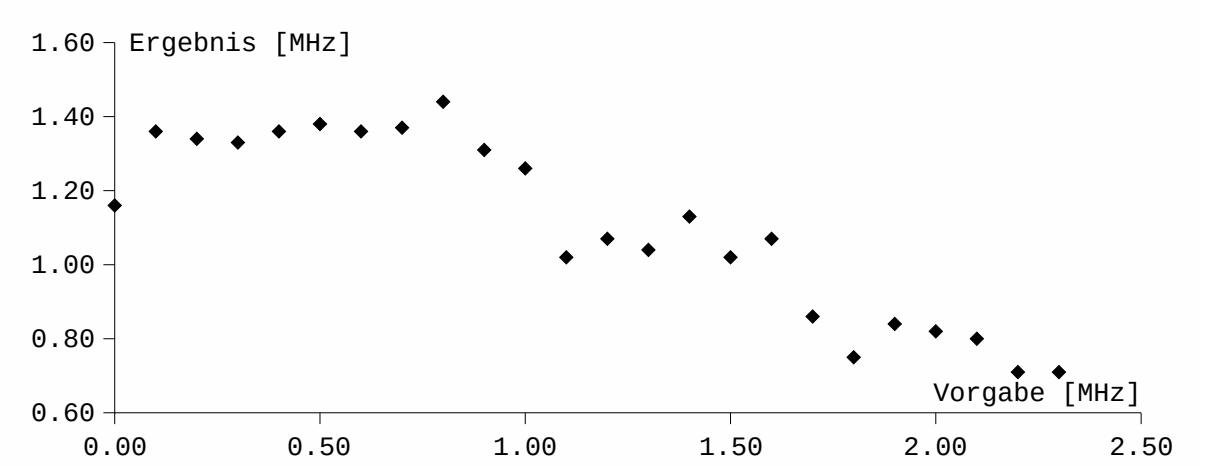


Bild 6.66: Einfluß von Taktvorgaben auf den kritischen Pfad der RTL-IDEA-Stufe

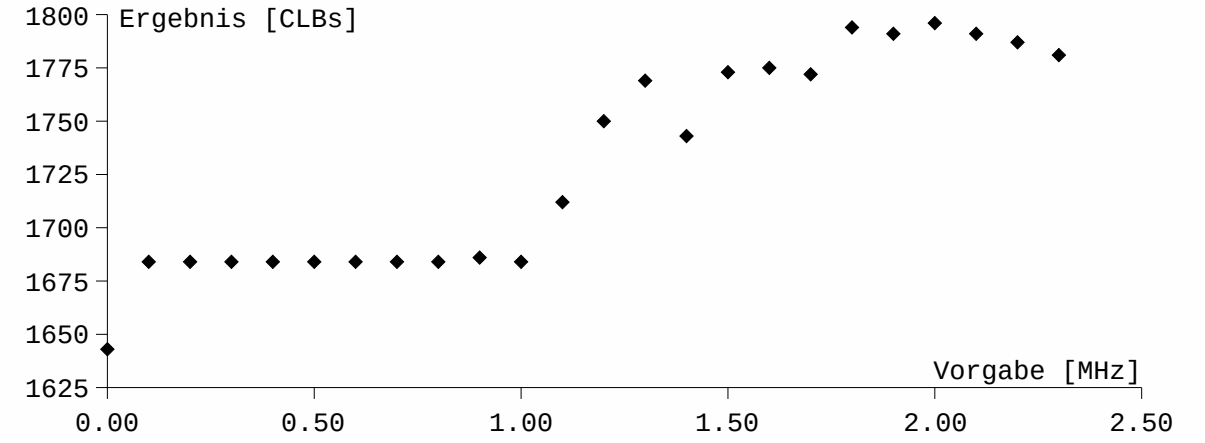


Bild 6.67: Einfluß von Taktvorgaben auf die Größe der RTL-IDEA-Stufe

High-Level-Synthese einer IDEA-Stufe: Die High-Level-Beschreibung der IDEA-Stufe besteht aus einem Modul mit einem `always`-Prozeß. In diesem Prozeß werden synchronisiert auf eine steigende Taktflanke die arithmetischen und logischen Verknüpfungen entsprechend Bild 6.61 in prozeduralen Zuweisungen ausgeführt und dabei die in eine Funktion ausgelagerte Multiplikation aufgerufen. Die Ausgangsdaten werden, impliziert durch die bei der High-Level-Synthese eingesetzte FSMD-Architektur, in Registern gespeichert, so daß bei Anreihung mehrerer Stufen eine Pipeline entsteht. Die Ergebnisse der mit verschiedenen Optionsbelegungen ausgeführten Syntheseläufe sind in Tabelle A.22 erfaßt, wobei wegen der variablen Schedule-Länge die Verarbeitungszeit angegeben wird, die das Produkt aus kombinatorischer Laufzeit (1 / Taktrate) und Taktanzahl ist.

Eine Unterscheidung dieser Ergebnisse nach der Flächenvorgabe in Bild 6.68 läßt keine Auswirkungen auf die Größe oder die Geschwindigkeit der Schaltung hervortreten. Ohne zielgerichtete Wirkung sind auch die Strukturoptimierungen (Bild 6.69) und die `compile`-Optionen (Bild 6.70).

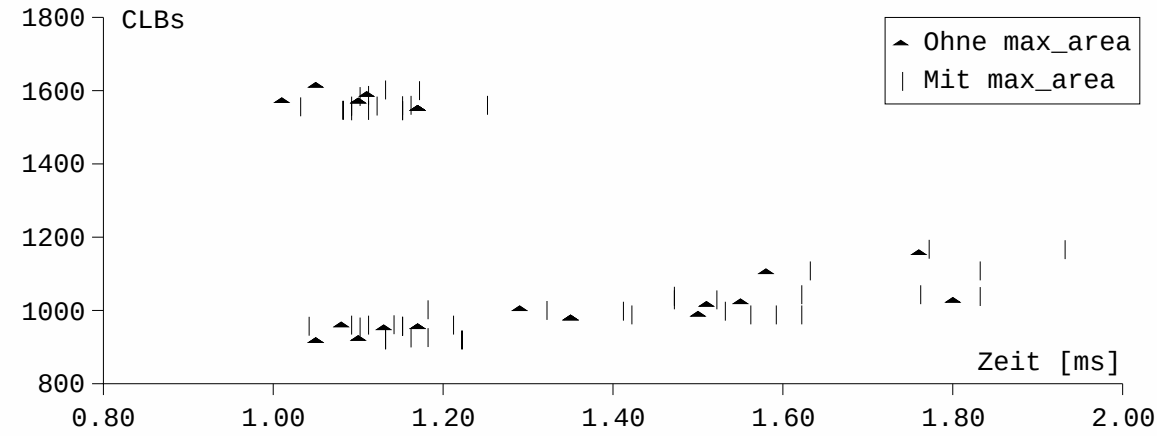


Bild 6.68: `set_max_area` nach der High-Level-Synthese der IDEA-Stufe

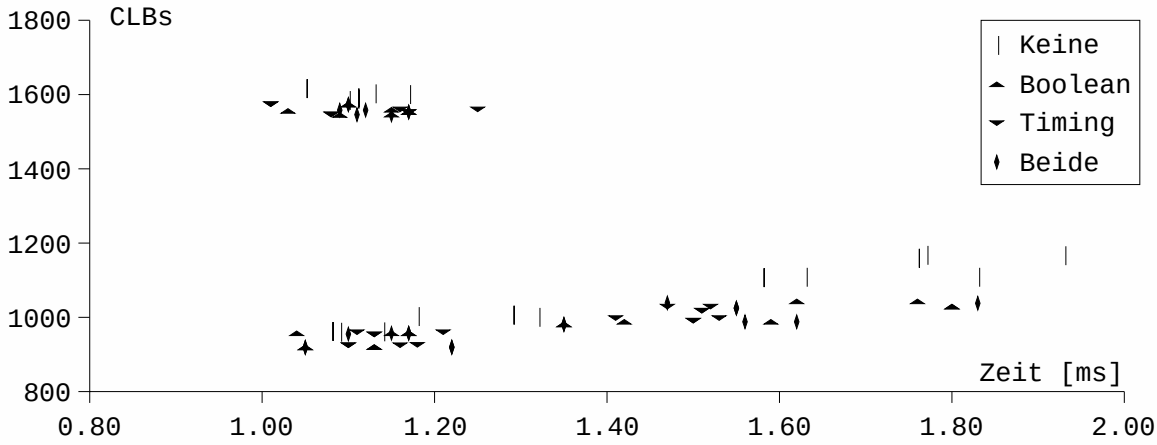


Bild 6.69: Strukturoptimierungen nach der High-Level-Synthese der IDEA-Stufe

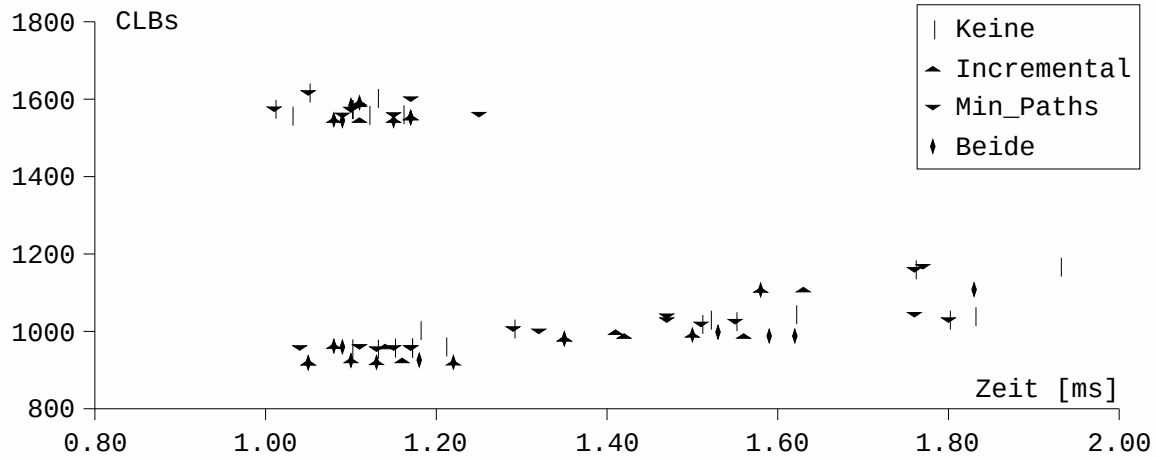


Bild 6.70: `compile`-Optionen nach der High-Level-Synthese der IDEA-Stufe

Die in Bild 6.71 markierten Taktvorgaben unterteilen das Ergebnisfeld in deutlich getrennte Bereiche, wobei mit 2MHz als Vorgabe kleine und gleichzeitig schnelle Schaltungen entstehen, während bei 1MHz und 3MHz entweder die Logikmenge oder die Verarbeitungszeit größer sind.

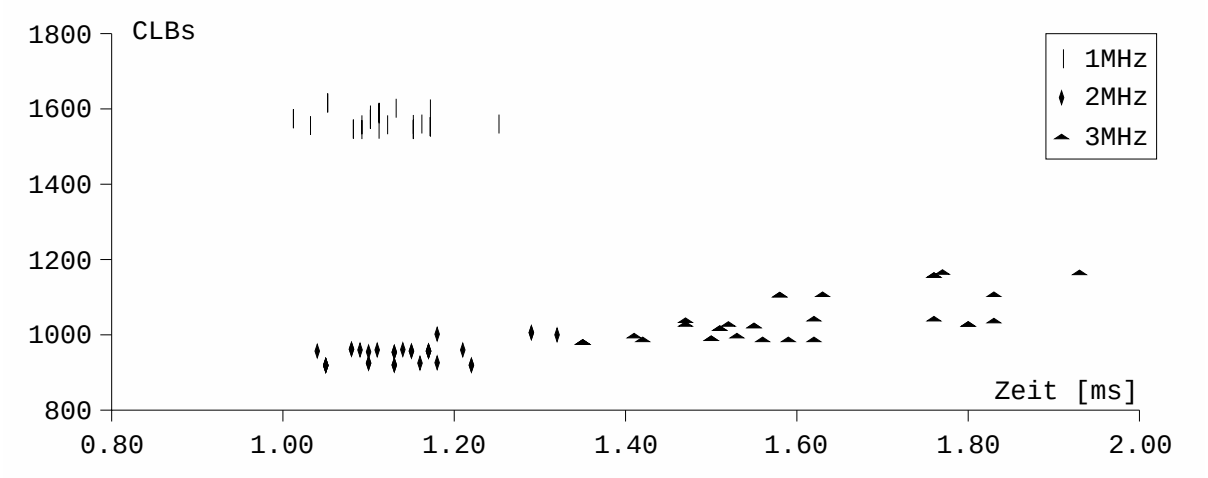


Bild 6.71: Taktvorgaben bei der High-Level-Synthese der IDEA-Stufe

Eine genauere Untersuchung des erkennbaren Einflusses der Taktvorgabe auf die Größe und die Verarbeitungszeit der IDEA-Stufe wurde mit Vorgabewerten von 0,5 bis 18,5MHz in Schritten zu 500kHz vorgenommen. Dabei wurden eine minimale Flächenvorgabe, beide Strukturoptimierungen und keine `compile`-Optionen benutzt. Tabelle A.23 faßt die dabei gewonnenen Daten zusammen.

Die Verzögerung im kritischen Pfad der Schaltung und damit die erreichte Taktrate folgt den Vorgabewerten fast linear zwischen 0,5 und 6MHz. Vorgabewerte über 6MHz produzieren durchweg langsamere Ergebnisse, die für Vorgaben bis 13MHz stark schwanken und sich darüber hinaus bei etwa 4MHz einpendeln (Bild 6.72).

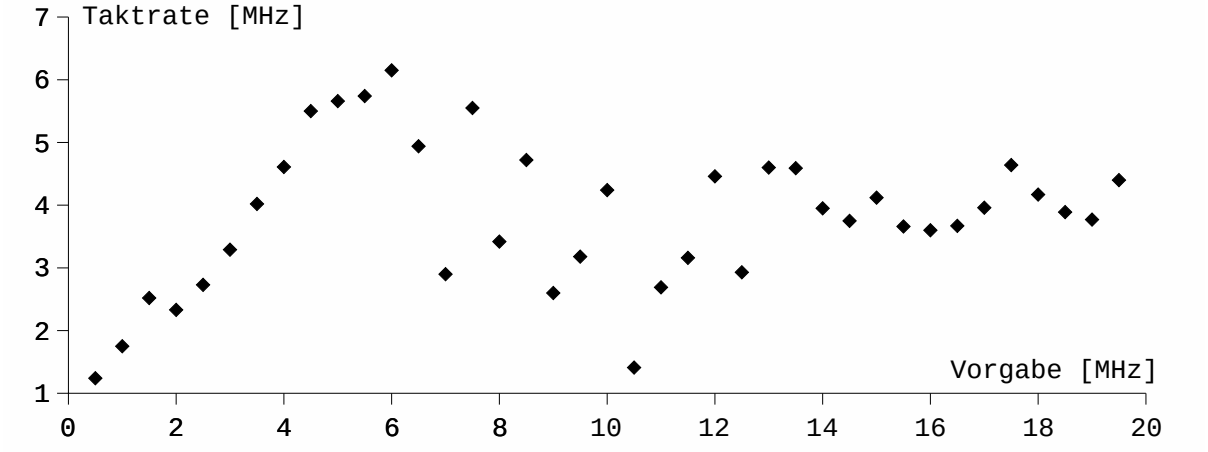


Bild 6.72: Einfluß von Taktvorgaben auf den kritischen Pfad der HL-IDEA-Stufe

Bei der in Bild 6.73 dargestellten Schaltungsgröße ist mit wachsenden Vorgaben zwischen 0,5 und 4MHz eine Verkleinerung feststellbar. Über 4MHz hinaus steigt

die benötigte Logikmenge an, ohne den bei 0,5 und 1MHz benötigten Höchstwert wieder zu erreichen.

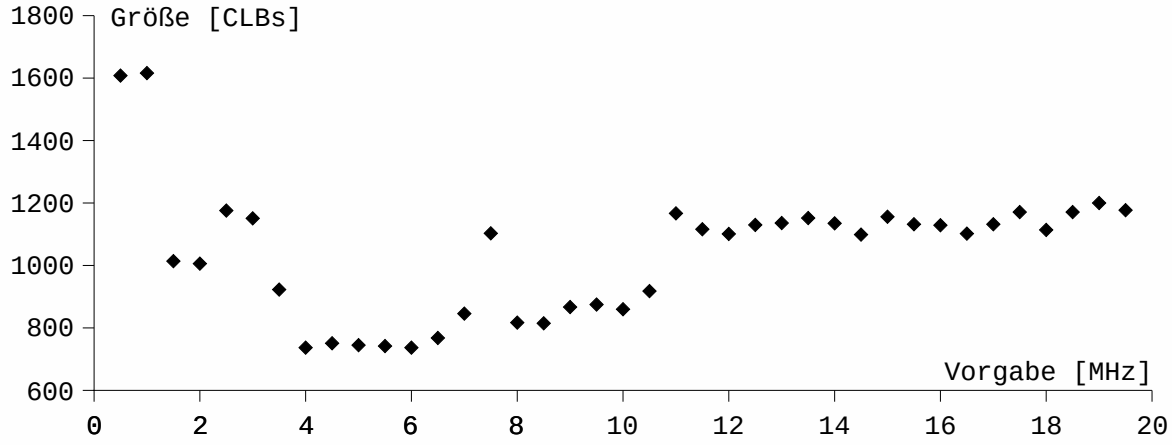


Bild 6.73: Einfluß von Taktvorgaben auf die Größe der HL-IDEA-Stufe

Die Verarbeitungszeit der Eingangsdaten in Bild 6.74 steigt, abgesehen von einigen Schwankungen bei Vorgaben zwischen 6 und 13MHz, bedingt durch die wachsende Anzahl der beim Scheduling festgelegten Verarbeitungstakte mit den Vorgabewerten an. Dies erklärt sich durch die bei jeder Registerstufe notwendigen Setup- und Hold-Zeiten, die zu den kombinatorischen Laufzeiten zwischen den Registern hinzukommen.

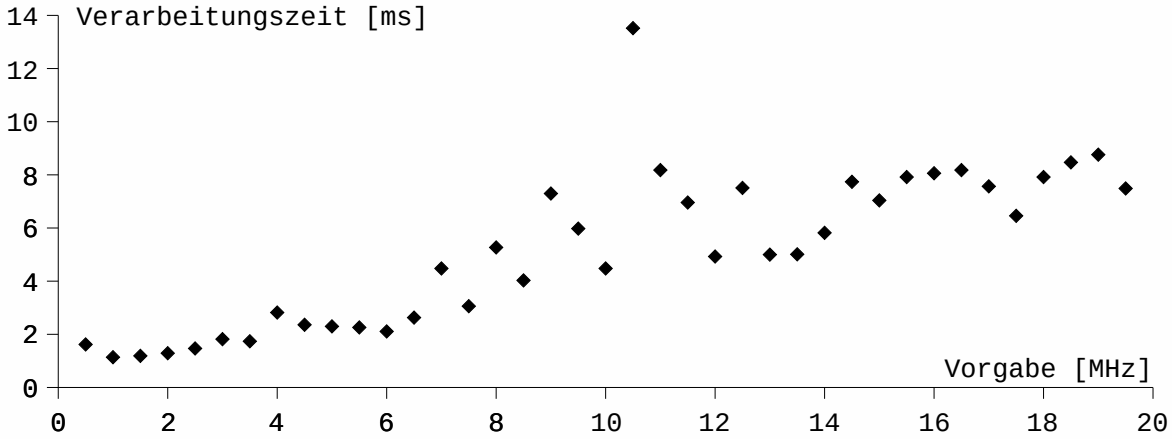


Bild 6.74: Einfluß von Taktvorgaben auf die HL-IDEA-Verarbeitungszeit

Zusammenfassung der IDEA-Synthese: In der Gegenüberstellung der RTL- und der High-Level-Syntheseergebnisse (Bild 6.75) für Größe und Verarbeitungszeit ist der wesentlich größere Ergebnisspielraum der High-Level-Synthese auffällig. Bei geringfügig höherer Verarbeitungszeit liefern die High-Level-Optimierungen von Scheduling und Allocation Ergebnisse, die nur etwa halb so groß sind wie die der RTL-Synthese. Werden um den Faktor 2 bis 3 höhere Laufzeiten akzeptiert, so sind noch kleinere Ergebnisse erreichbar.

Damit ist die High-Level-Synthese für diesen Entwurf im Vorteil gegenüber der RTL-Synthese. Bei vergleichbarer Entwurfsdauer ist ohne Modelländerungen

ein größerer Entwurfsspielraum vorhanden, in dem bei ähnlicher Geschwindigkeit weniger Logikressourcen benötigt werden. Der Entwurf ist damit flexibler an die Gegebenheiten des Umfeldes anpaßbar. Für den gesamten Datenpfad ist wegen der für kleinere Einzelstufen geringeren Verbindungsverzögerungen außerdem eine Verschiebung des Laufzeitvorteils zugunsten des High-Level-Entwurfs zu erwarten.

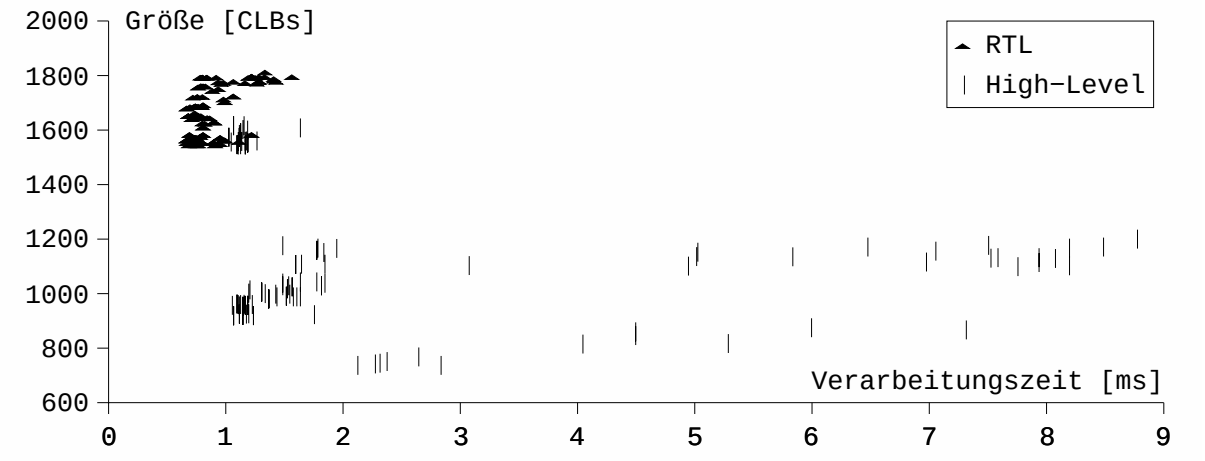


Bild 6.75: Ergebnisse der IDEA-Synthesemethoden

Die Auswirkungen der einzelnen Syntheseoptionen unterscheiden sich bei diesem Entwurf von den in vorangegangenen Abschnitten beobachteten Effekten, was teils auf den hohen Anteil arithmetischer Operatoren an der Logik zurückgeführt werden kann. Diese Operatoren werden in ihrer Implementierung aufgrund der Zeitvorgaben ausgewählt, bleiben aber von Flächen-, Struktur- und Übersetzungsoptimierungen ausgenommen. Die inkrementelle Übersetzung der Multiplikation Modulo 2^16+1 des RTL-Entwurfs verbessert aufgrund der mehrfachen Nutzung und Übersetzung dieses Moduls dafür die Ergebnisse unerwartet gut.

Taktvorgaben treten bei der IDEA-Synthese als durchgängig wirkungsvolle Syntheseoption hervor, bei der aber bereits durch kleine Variationen starke Schwankungen in Größe und Geschwindigkeit der Ergebnisse hervorgerufen werden können. Die Möglichkeit zur Überforderung der Optimierungen mit zu hohen Taktvorgaben und die daraus folgende Verschlechterung der Ergebnisse führt zur Notwendigkeit einer iterativen Bestimmung geeigneter Vorgabewerte.

6.3 Entwurfsklassifizierung und Syntheseauswahl

Bei den im vorangegangenen Abschnitt mit verschiedenen Synthesemethoden und -optionen untersuchten Entwürfen zeichneten sich Zusammenhänge zwischen dem jeweiligen Aufgabenprofil und der Eignung der Synthesemethoden sowie ihrer Optionen ab. Auf der Grundlage einer Klassifizierung der Entwürfe nach ihren aufgabenbedingten Eigenschaften werden diese Zusammenhänge zunächst weiter analysiert und für die Auswahl von Synthesemethoden verallgemeinert.

Anhand der Übersicht in Tabelle 6.2 über die Vorgaben an die Entwürfe werden zunächst die vor Entwurfsbeginn als Entscheidungsgrundlage nutzbaren

Eigenschaften vorgestellt, bevor sie für die beabsichtigte Klassifizierung weiterentwickelt werden.

	discount	DAR	MRISC	CardRead	IDEA
Taktung	14.32MHz	16MHz	maximal	1MHz	maximal
Zielhardware	minimal	minimal	4010XL-3	4013E-4	4052XL-3
I/O-Protokoll, Verlauf	Taktraster, zyklisch	Taktraster, datenabhängig	Taktraster, zyklisch	Handshake, datenabhängig	Durchlauf, zyklisch
Dominanz des Kontroll- oder Datenflusses	Kontrollfluß	Kontrollfluß	Datenfluß	Kontrollfluß	Datenfluß
Takt kritisch für Zielhardware	ja	ja	ja	nicht vorab entscheidbar	ja
Größe kritisch für Zielhardware	ja	ja	nicht vorab entscheidbar	nicht vorab entscheidbar	nicht vorab entscheidbar

Tabelle 6.2: Überblick über Vorgaben an die betrachteten Entwürfe

Bei der Taktung ist der genaue Wert an sich als Entscheidungskriterium ohne Bedeutung. Nur durch die Verbindung dieser Vorgabe mit der Zielhardware, die durch die Laufzeiten besteht und darüber entscheidet, ob die Taktvorgabe kritisch ist, besitzt sie Einfluß auf die Entwurfsentscheidungen. Die Laufzeiten können jedoch erst während der Implementierung abgeschätzt werden, wenn mit der gewählten Synthesemethode aus der Entwurfsbeschreibung die Komponenten und ihre Verbindungen generiert werden. Vor Entwurfsbeginn kann die Taktvorgabe nur dann als sicheres Entscheidungskriterium benutzt werden, wenn sie sich durch eine qualitative Vorgabe (z.B. „minimal“) für die Taktrate oder die Zielhardware als kritisch oder unkritisch für den Entwurf abzeichnet.

Ebenso ist eine genaue Vorgabe der Zielhardware erst während des Entwurfs als Entscheidungskriterium nutzbar, da sich erst dabei herausstellt, wieviele Ressourcen belegt werden und wie schnell diese arbeiten. Die angestrebte Nutzung als sicheres Entscheidungskriterium vor dem Entwurfsbeginn setzt wiederum qualitative Vorgaben für die Zielhardware voraus.

Damit ergeben sich anstelle der Anforderungen für die Taktrate und die Zielhardware als Entscheidungskriterien die Eigenschaften, ob die Taktrate und die Schaltungsgröße für die benutzte Zielhardware kritisch sind. Ist dies wegen konkreter Vorgaben für den Takt oder die Zielhardware nicht vorab entscheidbar, so wird vom ungünstigsten Fall für den Entwurf ausgegangen und eine kritische Einstufung gewählt. Die Kombination dieser Eigenschaften bietet gemäß der Vorgabemöglichkeiten in Tabelle 6.3 neun Möglichkeiten. Für reale Entwürfe bedeutungslose Vorgaben wie minimaler Takt oder maximal große und schnelle Zielhardware werden nicht berücksichtigt.

Takt		Zielhardware		
		konkrete Angabe	qualitative Angabe	
			minimal	egal, wird später passend gewählt
konkrete Angabe		Takt: (ja) Größe: (ja)	Takt: ja Größe: ja	Takt: nein Größe: nein
qualitative Angabe	maximal	Takt: ja Größe: (ja)	Takt: ja Größe: ja	Takt: ja Größe: nein
	egal, jeder Takt reicht	Takt: nein Größe: (ja)	Takt: nein Größe: ja	Takt: nein Größe: nein

Tabelle 6.3: Einstufungen für Takt und Größe vor Entwurfsbeginn

Die Anforderungen an die zeitliche Festlegung und den Verlauf von Ein- und Ausgaben sind in der angegebenen, qualitativen Form direkt verwendbar. Eine genauere Spezifikation mit Zeitangaben ist für Entscheidungen vor Entwurfsbeginn wegen fehlender Informationen zur Architektur und dem Verarbeitungsschema nicht notwendig. Ein sich zyklisch wiederholender oder von Daten abhängiger Verlauf der Ein- und Ausgaben ist dabei als Hilfe bei der Einschätzung des Daten- und Kontrollflusses anwendbar, die lediglich grob auf Basis der Entwurfsaufgabe möglich ist. Hierbei wird der Datenfluß, ohne formal eine präzise Metrik zu definieren, durch die zusammenhängenden Operationen auf Daten und deren Bit-Breite beeinflußt. Der Kontrollfluß wird durch die Menge an bedingten Operationen nach Abfrage von Datenwerten bestimmt. Die Dominanz von Daten- oder Kontrollfluß gibt an, ob der Entwurfsaufwand mehr aus der Verknüpfung von Daten oder mehr aus der Auswertung von Daten resultiert.

Neben den Vorgaben an die Entwürfe ist für diese vorab die Einschätzung der Komplexität möglich, die sich aus der Menge der Zustände im Kontrollfluß bzw. der Anzahl der Operationen im Datenfluß ableitet und als Eigenschaft Einfluß auf die Entwurfsdauer hat. Dabei wird hier nur zwischen geringer, mittlerer und hoher Komplexität unterschieden, je nachdem, ob die Zustände bzw. Operatoren noch vollständig (gering), teilweise (mittel) oder nicht (hoch) überschaubar sind.

Als Kriterien zur Auswahl einer Synthesemethode zu einer Entwurfsaufgabe werden auf der Basis der vorangegangen Betrachtungen folgende Eigenschaften gewählt:

1. Takteinstufung
2. Größeneinstufung
3. Art der Ein-/Ausgabesynchronisation (I/O-Protokoll)
4. Entwurfsschwerpunkt im Daten- oder Kontrollfluß
5. Komplexität des Daten- bzw. Kontrollflusses

Eine Klassifizierung von Entwürfen ist durch diese Eigenschaften und eine feste Menge von Ausprägungen möglich. Für die im vorigen Abschnitt betrachteten Entwürfe ergibt sich die Entwurfsklassifizierung nach Tabelle 6.4. In der Tabelle sind außerdem die Ergebnisse der drei Synthesemethoden eingetragen, so daß die Zusammenhänge zwischen dem Aufgabenprofil eines Entwurfs und der für ihn geeigneten Synthesemethode hervortreten.

	discount	DAR	MRISC	CardRead	IDEA
Takteinstufung (kritisch, unkritisch)	kritisch	kritisch	kritisch	kritisch	kritisch
Größeneinstufung (kritisch, unkritisch)	kritisch	kritisch	kritisch	kritisch	kritisch
I/O-Synchronisation (Taktraster, Handshake, keine)	Taktraster	Taktraster	Taktraster	Handshake	keine
Entwurfsschwerpunkt (Datenpfad, Controller)	Controller	Controller	Datenpfad	Controller	Datenpfad
Komplexität (gering, mittel, hoch)	gering	mittel	mittel	hoch	gering
RTL-Synthese	sehr gut	sehr gut	gut	gut	gut
Controllersynthese	gut	gut	-	sehr gut	-
High-Level-Synthese	schlecht	schlecht	sehr gut	-	sehr gut
Dauer RTL-Entwurf	1 Tag	1 Woche	1 Woche	9 Wochen	1/2 Woche
Dauer Controllerentwurf	1/2 Tag	1/2 Woche	-	3 Wochen	-
Dauer High-Level-Entwurf	3 Tage	1 Woche	1/2 Woche	-	1/2 Woche

Tabelle 6.4: Klassifizierung und Ergebnisse der betrachteten Entwurfsaufgaben

Anhand der Entwurfsklassifizierungen und der Syntheseergebnisse in Tabelle 6.4 wird deutlich, daß nur ein kleiner Teilraum des insgesamt möglichen Aufgabenraumes betrachtet wurde. Von den insgesamt möglichen 48 Kombinationen der Eigenschaften wurden nur 4 durch die untersuchten Entwürfe abgedeckt. So sind für Entwürfe mit unkritischer Takt- oder Größeneinstufung sowie für mehrere Kombinationen der I/O-Synchronisation und des Entwurfsschwerpunktes nur eingeschränkt Aussagen möglich, die auf dem Verhältnis der Ergebnisbereiche untereinander und den charakteristischen Eigenschaften der Synthesemethoden aufbauen. Das behandelte Forschungsgebiet kann durch weitere Untersuchungen noch vervollständigt werden und wird nicht durch einzelne Arbeiten wie diese erschöpfend behandelt werden.

In dem abgedeckten Teilraum sind aber genügend wichtige Entwurfsfälle enthalten, die dem beabsichtigten Ziel der Untersuchungen entsprechend die Auswahl einer geeigneten Synthesemethode in Fällen erlauben, in denen keine charakteristische Eigenschaft einer Synthesemethode die Entscheidung bestimmt.

Folgende Zusammenhänge sind zwischen den Anforderungen und der Eignung der Synthesemethoden erkennbar:

- Ist der Entwurf vom Schwerpunkt her ein Controller, so sind die RTL- und die Controllersynthese zur Implementierung geeignet, die High-Level-Synthese aber ungeeignet.
- Bei kleinen Controllern mit einer überschaubaren Zustandsmenge liefert die RTL-Synthese die besten Ergebnisse. Bei komplexen Controllern verlagert sich der Vorteil zur Controllersynthese, die weniger Entwurfszeit benötigt.
- Die High-Level-Synthese ist für Datenpfade besser geeignet als die RTL-Synthese, sofern deren Verarbeitungsschema wie in den untersuchten Entwürfen nicht an das I/O-Protokoll gebunden ist.

Aus dem Verhältnis der Ergebnisbereiche verschiedener Synthesemethoden für die unterschiedlichen Entwürfe des vorigen Abschnitts zeichnen sich außerdem einige Zusammenhänge für die Takt- und Größeneinstufung ab, die jedoch nicht an Entwürfen mit derartigen Anforderungen verifiziert wurden:

- Für Controller mit unkritischer Takt- und Größeneinstufung ist die RTL-Synthese wegen ihrer höheren Entwurfszeiten bereits bei kleinen Zustandsmengen weniger gut geeignet als die Controllersynthese.
- Ist bei einem Controller nur die Takt- oder Größeneinstufung kritisch, so sind für kleine Zustandsmengen mit gezielten Optimierungen bei der RTL-Synthese bessere Ergebnisse im kritischen Bereich zu erwarten als mit der Controllersynthese.
- Komplexe Controller sind unabhängig von Takt- und Größeneinstufung mit der Controllersynthese effizienter zu entwerfen als mit der RTL-Synthese.
- Unabhängig von der Takt- und Größeneinstufung bietet die High-Level-Synthese für Datenpfade günstigere Entwurfszeiten und -ergebnisse als die RTL-Synthese, sofern das Verarbeitungsschema nicht taktgebunden ist.
- Ist die Verarbeitung in einem Datenpfad an ein vorgegebenes Taktschema gebunden, so sind die High-Level-Optimierungen Scheduling und Allocation eventuell nicht durchführbar. Die RTL-Synthese hat in diesen Fällen den Vorteil, immer zu verwertbaren Ergebnissen zu führen.

Die Entwicklung der unterschiedlichen Entwurfsmodelle ergab zusätzlich:

- Trotz eines Entwurfsschwerpunktes im Datenpfad- oder Controllerbereich kann der jeweils andere Bereich bedeutend genug sein, um eine Aufteilung in Teilentwürfe für verschiedene Synthesemethoden für eine effiziente Lösung zu erfordern (Chipkartenleser: RTL-Entwurf der BCD-Addition, Controllerentwurf der übrigen Funktionalität).
- Die Controllersynthese ist für die Konstruktion von Datenpfaden ungeeignet.

Für die Auswahl einer Synthesemethode ist neben diesen Zusammenhängen die Gewichtung der Entscheidungskriterien untereinander wichtig, die der Abfolge in dieser Liste entspricht:

1. Entwurfsschwerpunkt (Controller oder Datenpfad)

2. Bindung zwischen I/O-Synchronisation und Verarbeitung
3. Komplexität
4. Takt- und Größeneinstufung

Erfordert ein Entwurf keine der in Abschnitt 6.1 genannten charakteristischen Eigenschaften einer Synthesemethode, so wird zur Auswahl anhand des Entwurfsschwerpunktes die jeweils ungeeignete Synthesemethode bestimmt oder eine Aufteilung in Teilentwürfe vorgenommen, für welche der Auswahlprozeß neu beginnt. Controller sind für die High-Level-Synthese ungeeignet, Datenpfade für die Controllersynthese. Die RTL-Synthese ist für beide Entwurfsschwerpunkte gut geeignet, aber oft weniger gut als eine der anderen beiden Synthesemethoden.

Anhand der Bindung zwischen der I/O-Synchronisation des Entwurfs(-teils) und der Verarbeitung erfolgt die weitere Einschränkung der Synthesemethoden. Ist bei Datenpfaden eine feste Taktbindung vorhanden, so besteht das Risiko mit der High-Level-Synthese zu keinem Ergebnis zu kommen. Für solche Datenpfade ist nur die RTL-Synthese uneingeschränkt geeignet.

Besitzt der Entwurf eine hohe Komplexität, so erfordert ein RTL-Entwurf eine hohe Entwurfszeit und ist nicht so effizient implementier- und optimierbar wie mit den anderen beiden Methoden, sofern diese zum Zeitpunkt dieser Entscheidung noch in Frage kommen. Bei geringer und mittlerer Komplexität bleibt sie jedoch in der Auswahl.

Sind Takt und Größe unkritisch, so sind statt der RTL-Synthese verbliebene Alternativen zu bevorzugen.

6.4 Auswahl von Syntheseoptionen

Nach der Auswahl einer geeigneten Synthesemethode zu einem Aufgabenprofil, die im vorangegangenen Abschnitt beschrieben wurde, sind die Optionen der Synthesemethoden für bestmögliche Ergebnisse zu wählen. Die in Abschnitt 6.2 vorgestellten empirischen Untersuchungen liefern hierfür eine solide Grundlage, anhand derer feste Vorbelegungen und Variationsarten der Optionen bestimmbar sind. Einen Überblick über die Auswirkungen der Optionen in der RTL-Synthesephase der drei Synthesemethoden für die untersuchten Entwürfe gibt Tabelle 6.5.

Entwurf	Flächen-vorgabe (minimal)	Strukturoptimierungen		compile-Optionen		Taktvor-gabe
		boolean	timing	increment.	min_paths	
discount (RTL)	unklar	kleiner & schneller	kleiner & schneller	unklar	unklar	größer & schneller
discount (High-Level)	unklar	kleiner & langsamer	kleiner & langsamer	größer & langsamer	unklar	nicht variierbar
discount (Controller)	unklar	kleiner	kleiner	langsamer	keine	schneller

Tabelle 6.5: Auswirkungen der RTL-Syntheseoptionen auf die Ergebnisse

Entwurf	Flächen-vorgabe (minimal)	Strukturoptimierungen		compile-Optionen		Taktvor-gabe
		boolean	timing	increment.	min_paths	
DAR (RTL)	unklar	kleiner	kleiner	unklar	unklar	schneller
DAR (High-Level)	unklar	kleiner & schneller	kleiner & schneller	langsamer	unklar	nicht variierbar
DAR (Controller)	unklar	kleiner	kleiner	unklar	keine	schneller
MRISC (RTL)	unklar	kleiner	kleiner	unklar	keine	größer & schneller
MRISC (High-Level)	unklar	keine	keine	größer	keine	größer & schneller
CardRead (RTL)	größer & schneller	kleiner & langsamer	kleiner & langsamer	unklar	keine	kleiner & schneller
CardRead (Controller)	schneller	kleiner & langsamer	kleiner & langsamer	unklar	keine	kleiner & schneller
IDEA (RTL)	unklar	unklar	unklar	kleiner	unklar	größer & schneller
IDEA (High-Level)	unklar	unklar	unklar	unklar	unklar	kleiner & langsamer

Tabelle 6.5: Auswirkungen der RTL-Syntheseoptionen auf die Ergebnisse

Auf der Grundlage dieser Übersicht und den in Abschnitt 6.2 erwähnten Besonderheiten lassen sich mehrere Leitlinien für die Auswahl der RTL-Syntheseoptionen aufstellen:

- Besonders wichtig sind die Taktvorgaben und die Strukturoptimierungen, da diese den deutlichsten Einfluß auf die Ergebnisqualität besitzen. Die anderen Optionen können wegen ihres unklaren und geringen Einflusses bis zur Abschlußoptimierung des Entwurfs vernachlässigt werden. Ein Ausprobieren ihrer Variationsmöglichkeiten kann dann die bereits erzielten Ergebnisse vielleicht noch verbessern.
- Die Strukturoptimierungen sollten generell aktiviert werden. Sind die damit erreichten Ergebnisse zu langsam, sollte durch ihre testweise Deaktivierung überprüft werden, ob sich so die Geschwindigkeit steigern läßt.
- Zur Bestimmung einer geeigneten Taktvorgabe sollte zunächst eine Synthese ohne Taktspezifikation (RTL-Synthese) oder mit einem sehr niedrigen Takt erfolgen (High-Level-Synthese). Die so erzielte Geschwindigkeit ist in einem weiteren Lauf als Spezifikation einzusetzen. Werden auf diese Weise bessere Ergebnisse erzielt, so kann durch eine schrittweise Steigerung des Vorgabewertes das Optimierungsmaximum gesucht werden.
- Sind geeignete Einstellungen für die Strukturoptimierungen und den Takt gewählt, so bietet die Übersetzung mit `incremental_map` zusätzliches

Optimierungspotential. Diese Option wirkt aber nur dann verbessernd, wenn eine benutzte Unterinstanz bereits ohne diese Option vorübersetzt wurde.

- Abschließend kann durch (De-)Aktivierung der Flächenvorgabe bzw. der `min_paths`-Option geprüft werden, ob noch weiteres Optimierungspotential besteht.

Für die der RTL-Synthese vorgeschaltete Controller- bzw. High-Level-Synthese wurden zu wenige Optionen geprüft und zu unklare Ergebnisse erzielt, um daraus Schlußfolgerungen ziehen zu können. Hier wären noch weitere Untersuchungen nötig, die den abgesteckten Rahmen dieser Arbeit jedoch sprengen würden.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurden die RTL-, die High-Level- und die Controllersynthese als Methoden des automatisierten Schaltungsentwurfs betrachtet, in ihren Möglichkeiten verglichen und anhand verschiedener Schaltungen experimentell auf ihre jeweilige Anwendungseignung hin untersucht.

Dabei wurden zunächst allgemeine Grundlagen dieser Methoden vorgestellt und elementare Begriffe der Schaltungssynthese durch Erläuterungen festgelegt. Die Sprachen zur Hardwarebeschreibung wurden ebenso wie die Werkzeuge zur Simulation und zur Synthese vorbereitend zur Vorstellung der Entwurfsabläufe eingeführt. Auf die Konzepte der in den Syntheseuntersuchungen dieser Arbeit verwendeten Sprachen Verilog und Protocol-Compiler-HDL wurde sowohl einzeln als auch im Vergleich eingegangen. Als Abschluß des Grundlagenbereichs wurde die Quantifizierung und Bewertung von Syntheseergebnissen bei Verwendung von Xilinx-FPGAs betrachtet. Mit dem experimentellen Nachweis einer signifikanten Korrelation zwischen der Anzahl von FPGA-CLBs und der Gatteranzahl maskenprogrammierter ASICs (Gate-Count) wurde das CLB-Größenmaß als Bewertungsmöglichkeit von Syntheseergebnissen bestätigt.

Darauf aufbauend wurden die Möglichkeiten der RTL-, High-Level- und Controllersynthese hinsichtlich der Modellierung mit Hardwarebeschreibungssprachen und der Umsetzung der Modelle mit den Synthesewerkzeugen näher vorgestellt. Hierbei wurden zu den aus Spezifikationen und Dokumentationen ersichtlichen Informationen zusätzlich auch die im Rahmen mehrerer Entwurfsprojekte gesammelten praktischen Erfahrungen mit berücksichtigt.

Bei der RTL-Synthese wurde dazu nach der Beschreibung der verwendbaren Verilog-Konstrukte anhand von Beispielen ihre Zusammensetzung zu Modellen erläutert. In diesem Kontext wurde auf Fehlermöglichkeiten bei der Modellierung hingewiesen, zu deren Vermeidung Hinweise aus eigenen Entwurfserfahrungen und anderen Quellen gegeben wurden. Ein weiterer Schwerpunkt lag auf den Optimierungen während der Synthese und ihren Einflußmöglichkeiten im Vergleich zu den Optimierungen bei der Modellierung.

Die High-Level-Modellierung wurde sowohl in ihren Erweiterungen als auch in ihren Einschränkungen im Vergleich zur RTL-Modellierung eingeführt. Die wesentlichen Arbeitsschritte der High-Level-Synthese wurden unterstützt durch Beispiele beschrieben und in ihren Optimierungsmöglichkeiten bei der Abbildung auf eine FSMD-Architektur charakterisiert.

Die Controllersynthese mit dem Protocol-Compiler wurde als Zusatz zu den bereits präsentierten Grundlagen der Protokolldefinition um Einzelheiten der verschiedenen Definitionselemente ergänzt. Außerdem wurden die verschiedenen Zustandscodierungen der Protokollsynthese und die Controllerarchitekturen der Codeerzeugung beschrieben. Die Optimierungsmöglichkeiten bei der Modellierung und bei der Synthese bestimmen wie bei den anderen beiden Synthesemethoden die abschließende Betrachtung der Möglichkeiten der Controllersynthese.

Aus den Grundlagen- und Detailbetrachtungen dieser drei Synthesemethoden wird die bisher unzureichend beantwortete Fragestellung nach der Eignung der Methoden zur Lösung von Entwurfsaufgaben deutlich. Diese Frage bildet den Kern der vorliegenden Arbeit. Sie wird beantwortet durch eine eingehende Analyse der Sprach- und Werkzeugeigenschaften und durch zahlreiche praktische Entwurfsexperimente. Die angewandte Vorgehensweise, die erzielten Ergebnisse, die Auswirkungen auf Forschung und Lehre an der Abteilung E.I.S. bzw. die Protocol-Compiler-Entwicklung bei Synopsys werden nachfolgend zusammenfassend erläutert.

7.1 Analyse der Spracheigenschaften und -möglichkeiten

Bei der Suche nach der besten Synthesemethode wurden die Hardwarebeschreibungssprachen in ihren Eigenschaften, Möglichkeiten und werkzeugbedingten Einschränkungen analysiert und verglichen. Dazu wurden die im Grundlagenbereich vorgestellten Sprach- und Entwurfseigenschaften verfeinert und den durch Entwurfsaufgaben geforderten Eigenschaften gegenübergestellt.

Darauf aufbauend wurde eine Aufteilung der bei einer Synthesemethode nutzbaren Spracheigenschaften in zwei Bereiche erarbeitet: in charakteristische Eigenschaften, die nur bei einer Methode vorhanden sind und diese auszeichnen (z.B. bidirektionale Kommunikation bei der RTL-Synthese), und in allgemeine Eigenschaften, die bei mehreren Sprachen bzw. Methoden vorhanden sind (wie die Instanzierung von RTL-Komponenten).

Durch diese Aufteilung wird eine erste Auswahlhilfe für Synthesemethoden zu Entwurfsproblemen gegeben. Erfordert ein Entwurf eine charakteristische Eigenschaft einer Synthesemethode, so sind die Entwurfssprache bzw. -methode bereits durch die Aufgabe festgelegt. Auf ähnliche Weise wird mit den allgemeinen Eigenschaften die Auswahl an nutzbaren Methoden reduziert, wenn ein Entwurf eine Eigenschaft erfordert, die nicht bei allen Methoden vorhanden ist.

Es wird aber deutlich, daß eine rein auf Eigenschaften bzw. Möglichkeiten der Eingabesprachen und Synthesemethoden basierende Auswahl nicht ausreicht, um für alle Entwurfsaufgaben die jeweils beste Methode zu wählen. Es wurden daher praktische Syntheseexperimente durchgeführt, deren Ergebnisse eine abgestufte Auswahl zwischen Methoden erlauben, wenn charakteristische und allgemeine Eigenschaften keine eindeutige Entscheidung ermöglichen.

7.2 Syntheseexperimente und verfeinerte Auswahl

Als Versuchsobjekte für die praktischen Syntheseexperimente wurden Entwürfe aus Praktikumsaufgaben und Diplomarbeiten verwendet, die in vielen Aspekten repräsentativ für typische Industrie- und Forschungsprojekte sind. Für diese lagen teils alternative Lösungen in einer Methode vor und teils alternative Lösungen in mehreren Synthesemethoden (z.B. für Display-Controller oder MRISC-Prozessor). Neben den speziellen Optimierungsoptionen der High-Level- und der Controllersynthese wurden im gemeinsamen RTL-Synthesepfad aller drei

untersuchten Methoden die RTL-Optimierungen in verschiedenen Kombinationen angewendet.

Die erzielten Ergebnisse bilden Lösungsräume, die sich in ihrer Form und ihrer Lage zueinander je nach Entwurf und Methode unterscheiden. Die Einflüsse der Modellierung, der Synthesemethoden und der Optimierungsoptionen auf die Ergebnisse wurden über alle Entwürfe hinweg betrachtet und analysiert. Mittels dieser Untersuchungen wurde eine Klassifizierung für künftige Entwürfe entwickelt, anhand derer klare Abhängigkeiten zwischen den Anforderungen und den erreichbaren Ergebnissen eines Entwurfs erkennbar sind. Die Klassifizierung erfolgt nach:

1. Entwurfsschwerpunkt (Controller oder Datenpfad)
2. Bindung zwischen I/O-Synchronisation und Verarbeitung
3. Komplexität
4. Takt- und Größeneinstufung

Ausgehend von dieser Klassifizierung wurden Richtlinien für die Auswahl einer Synthesemethode zu einer Entwurfsaufgabe formuliert, welche die Ergebnisse in Größe, Geschwindigkeit und Entwurfsdauer gezielt optimieren. Weiterhin wurden Hinweise zur Entwurfspartitionierung gegeben, die eine Anpassung von Entwürfen mit breiten Anforderungsspektren an die Möglichkeiten mehrerer parallel eingesetzter Synthesemethoden zulassen.

Durch diese Auswahlrichtlinien und Partitionierungshinweise werden zum einen für neue Entwürfen die bereits beobachteten Effekte und zu erfüllenden Anforderungen besser vorhersagbar. Zum anderen sind aber auch die Probleme einiger abgeschlossener Entwurfsprojekte nachvollziehbar, beispielsweise bei der High-Level-Synthese in [Buch97], die wegen des eng vorgegebenen Taktschemas die Auslagerung wesentlicher Datenpfadelemente in RTL-Bibliotheken erforderte und so letztlich nicht effektiver war, als eine reine RTL-Synthese.

Bei den Untersuchungen wurden zudem für die Optionen des gemeinsamen RTL-Synthesepfades Einflüsse auf die Ergebnisqualität festgestellt, die aufgrund der Dokumentation des Design-Compilers in ihrer Art und ihrem Ausmaß vorher anders eingeschätzt wurden. Insbesondere der Einfluß von Taktvorgaben auf die Geschwindigkeit der Ergebnisse entspricht selten der erwarteten linearen Kennlinie. Zur Bestimmung einer optimalen Ergebnisgeschwindigkeit erweist sich eine iterative Bestimmung der Taktvorgabe als bestes Vorgehen. Dies wird an den Untersuchungen des MRISC-Prozessors und des IDEA-Datenpfades deutlich, bei denen eine zu hoch angesetzte Taktvorgabe langsamere Ergebnisse zur Folge hat, als eine iterativ bestimmte maximale Taktvorgabe (*clock overstraining*).

Für die Strukturoptimierungen zeigt sich ebenfalls ein starker Einfluß, wobei Boolesche Optimierungen mit strukturiertem Ansatz (`-structure true`) und mit Timing-Schwerpunkt (`-timing true`) bei nur geringen Einbußen in der Geschwindigkeit signifikante Größeneinsparungen erbringen.

Die entwickelten Auswahlkriterien bzw. -richtlinien wirken sich ebenso wie die Hinweise zur Modellierung und Optimierung außerhalb dieser Arbeit aus, worauf in den nachfolgenden Abschnitten weiter eingegangen wird.

7.3 Einfluß auf Entwurfsprojekte sowie Forschung und Lehre

Die in dieser Arbeit gewonnenen Ergebnisse und Erfahrungen haben auf viele Projekte Einfluß ausgeübt, der sich in Forschung und Lehre der Abteilung E.I.S. besonders deutlich nachzeichnen läßt.

Die Ausarbeitung von möglichst einfachen Modellierungsrichtlinien sowie von stabilen Entwurfsabläufen bereitete beispielsweise den Weg für den Einsatz der RTL-Synthese im Semi-Custom-Entwurfspraktikum. Teile des Vorlesungsskriptes [Golze99] sowie die Leitfäden zu den direkt oder indirekt mit Schaltungssynthese arbeitenden Praktika sind von den in dieser Arbeit gewonnenen Erkenntnissen maßgeblich beeinflußt.

In Studien- und Diplomarbeiten wurden diese Modellierungsrichtlinien und Entwurfsabläufe an verschiedenen nichttrivialen Entwürfen erprobt und zum Teil verfeinert [Kinder99] [Roggen99] [Friedr98] [Buch97]. Hierdurch wurde eine hohe Praxisnähe und -tauglichkeit der Syntheseanwendung erreicht, die sich zum Teil in den jeweils erfolgreich ausgeführten Post-Layout-Simulationen wiederspiegelt.

Die im Vergleich zu Studien- und Diplomarbeiten einfacheren Entwürfe des Semi-Custom-Praktikums ermöglichen durch den abschließenden Test in einer Hardwareumgebung darüber hinaus die Verifikation eines Entwurfsablaufs bis hin zum realen Einsatz der Schaltung. Beispiele für solche zusätzlich unter realen Bedingungen getestete Entwürfe sind der Bildschirmcontroller, der Digital-Audio-Receiver, der MRISC-Prozessor und der Chipkartenleser. Dies zeigt nicht nur die Anwendbarkeit der zugrunde liegenden Modellierungen und Entwurfsabläufe in der Praxis, sondern bildet gleichzeitig den Übergang zu Forschungsarbeiten am Protocol-Compiler.

Einer der ersten mit dem Protocol-Compiler bis hin zum realen Schaltungstest durchgeführten Entwürfe basiert auf der Praktikumsaufgabe des Digital-Audio-Receivers (DAR) [Blinze97] und diente als Beta-Test des Protocol-Compilers. Die erfolgreiche Umsetzung des zeit- und ressourcenkritischen Entwurfes mit dem neuen Werkzeug wurde auf nationalen und internationalen Fachkonferenzen mit Interesse aufgenommen [Blinze98] [BlHoGo98]. Die Vorstellung und praktische Vorführung auf der 35. Design Automation Conference bildete einen Höhepunkt dieses Teilprojektes [HolBli98].

Die systematischen Vergleiche der RTL-, der High-Level- und der Controller-synthese sowie die dabei gewonnenen Ergebnisse erweisen sich nicht nur für diese Arbeit und andere Veröffentlichungen von Bedeutung [Blinze99B] [Blinze00]. Sie werden auch in anderen Entwurfsprojekten verwertet, z.B. bei der optimierten Umsetzung von Statechart-Entwürfen für FPGAs oder für eine modifizierte Variante des ST7-Prozessors im Bereich der Home-Automation.

Für jetzt noch nicht abzusehende, neu zu beginnende Projekte ist der größte Nutzen zu erwarten. Diese können schon in der Planungsphase alle Ergebnisse dieser Arbeit voll ausschöpfen und somit die aufwendigen Korrekturen von vornherein ausschließen, die unter anderem in die hier aufgestellten Kriterien und Regeln eingeflossen sind.

7.4 Einfluß auf die Weiterentwicklung des Protocol-Compilers

Für die Entwicklung des Protocol-Compilers konnten im Rahmen der Beta-Tests und als Nebeneffekt der Untersuchungen dieser Arbeit mehrfach Anregungen gegeben werden.

So wurden mehrere kritische Fehler in der Verilog-Codeerzeugung und in der Protokolloptimierung entdeckt, die noch vor der Markteinführung der betroffenen Versionen korrigiert werden konnten.

Für die Codeerzeugung und hier insbesondere die Controllerarchitekturen wurden mehrere Verbesserungsvorschläge eingebracht, die neben einer höheren Ergebnisqualität nach den RTL-Optimierungen auf die vereinfachte Anbindung bidirektionaler Signale und die Nutzung festverdrahteter FPGA-Initialisierungen abzielen. Diese Vorschläge sind zum Teil in aktuellen Versionen berücksichtigt.

7.5 Ausblick

Der mit dem Abschluß der hier dokumentierten Arbeiten erreichte Zustand ist trotz seiner Bedeutung für die bestmögliche Anwendung der benutzten Synthesemethoden und -werkzeuge wegen der beständigen Veränderungen in diesem Forschungsgebiet kein Endpunkt mit dauerhafter Gültigkeit.

So sind die erreichten Ergebnisse für diese Methoden und Werkzeuge durch Untersuchungen an anderen Entwürfen noch verfeinerbar. Die Betrachtung von mehr als einem Werkzeug pro Methode und weiteren Zieltechnologien würde zur Allgemeingültigkeit der Ergebnisse und Aussagen beitragen und so die Gefahr verringern, lediglich in Nischenbereichen gültige Zusammenhänge untersucht zu haben.

Für neu zu beginnende Entwurfsprojekte ist diese Arbeit eine feste Grundlage, die Entscheidungen zur Verbesserung der Entwurfsqualität und zur Vermeidung von Fehlern zu treffen erlaubt. Vergleichbar dem TOOBSIE-Projekt [GoBCSW95], das Grundlagen der Modellierungstechnik zu dieser Arbeit beisteuerte, wurden Kenntnisse und Methoden erarbeitet, die als Standards für zukünftige Entwurfsprojekte und Forschungsvorhaben dienen werden. Direkt oder indirekt werden so viele der hier vorgestellten Richtlinien und Hinweise (wieder-)verwendet werden.

Die fortlaufende Verbesserung bestehender Werkzeuge sowie die Entstehung von neuen Methoden, Werkzeugen und Technologien werden in Zukunft die im Kontext dieser Arbeit beantwortete Frage nach der bestmöglichen Synthese für ein Entwurfsproblem immer wieder neu aufwerfen. Zu ihrer Beantwortung wird wie in dieser Arbeit eine Analyse der Eigenschaften und die praktische Untersuchung von Entwürfen für die Synthesemethoden notwendig sein.

Literatur

[Ascom98] Ascom Systec Ltd.; IDEA Encryption Algorithm; ISO standard 9979, 1998.

[BlHoGo98] Blinzer, P., Holtmann, U., Golze, U.; Entwurf von Controller-Schaltungen für Kommunikationsprotokolle mit dem Protocol-Compiler von Synopsys; GI / ITG / GMM Workshop “Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen”, Paderborn, März 1998.

[Blinze96] Blinzer, P.; Move-RISC-Prozessor; VLSI-Praktikum für Semi-Custom-Chips SS1996, TU Braunschweig, Abteilung E.I.S., 1996.

[Blinze97] Blinzer, P.; Design of a Digital Audio Receiver in a VLSI Lab; Electronic Circuits and Systems Conference, Bratislava, Slowakei, September 1997.

[Blinze98] Blinzer, P.; Using the Protocol Compiler for a critical FPGA Design; DATE ‘98, ESNUG , Paris, France, February 1998.

[Blinze99A] Blinzer, P.; Chipkarten-Auswertung mit einem FPGA; VLSI-Praktikum für Semi-Custom-Chips SS1999, TU Braunschweig, Abteilung E.I.S., 1999.

[Blinze99B] Blinzer, P.; Methoden und Werkzeuge der Schaltungssynthese; 9. E.I.S.-Workshop, Darmstadt, September 1999.

[Blinze00] Blinzer, P.; Selecting the best Synthesis Method in Chip Design; DATE 2000, ESNUG, Paris, France, March 2000.

[Buch97] Buchheim, K.; Verhaltenssynthese eines Mikrocontrollers; Technische Universität Braunschweig, Abteilung E.I.S., Diplomarbeit, 1997.

[CADrep96] CAD report; VHDL vs. Verilog: still a horse race -- new figures from EDAC reveal the two HDLs are about equally popular; Computer Aided Design report, August 1996, CAD/CAM Publishing Inc., 1996.

[EckHof92] Ecker, W. und Hofmeister M.; The Design Cube - A Model for VHDL Designflow Representation; European Design Automation Conference, Hamburg, Germany, 1992.

[Friedr98] Friedrichs, A.; Verhaltenssynthese im Schaltungsentwurf mit dem Behaviour Compiler im Vergleich mit etablierten Entwurfsmethoden; Technische Universität Braunschweig, Abteilung E.I.S., Diplomarbeit, 1998.

[GajKuh83] Gajski, D. und Kuhn, R.; Guest Editors Introduction: New VLSI Tools; IEEE Computer, volume 16, number 12, December 1983.

[Gajski88] Gajski, D.; Silicon compilation; Bibliography; Addison-Wesley Pubslishing, 1988.

[Gajski97] Gajski, D.; Principles of Digital Design; Prentice Hall, 1997.

[Gerez99] Gerez, Sabih H.; Algorithms for VLSI Design;Wiley & Sons, 1999.

[Golze99] Golze, U.; Einführung in den VLSI-Entwurf; TU Braunschweig, Abteilung E.I.S., Vorlesungsskript, 1999.

[GoBCSW95] Golze, U., Blinzer, P., Cochlovius, E., Schäfers, M., Wachsmann, K.-P.; VLSI-Entwurf eines RISC-Prozessors; Vieweg 1995.

[HLSynt89] Benchmarks set of Fourth International Workshop on High-Level Synthesis (HLSW-89); Kennebunkport, USA, October 1989.

[HLSynt91] Benchmarks set of Fifth International Workshop on High-Level Synthesis (HLSW-91); Bühlerrode, Germany, 1991.

[HLSynt92] Benchmarks set of Sixth International Workshop on High-Level Synthesis (HLSW-92); Laguna Niguel, CA, USA, November 1992.

[HLSynt95] Benchmarks set of Seventh International Workshop on High-Level Synthesis (HLSW-94); Niagara On-the-Lake, Ontario, Canada, May 1994.

[HolBli98] Holtmann, U., Blinzer, P.; Design of a SPDIF Receiver using Protocol Compiler; 35th Design Automation Conference, San Francisco, USA, Juni 1998.

[Kinder99] Kinder, S.; Modellierung einer IrDA-Geräteschnittstelle mit dem Synopsys Protocol-Compiler; Technische Universität Braunschweig, Abteilung E.I.S., Diplomarbeit, 1999.

[LGSynt89] Benchmarks set for the International Workshop on Logic Synthesis 1989 (IWLS-89); Microelectronics Center of North Carolina, USA, May 1989.

[LGSynt91] Benchmarks set for the International Workshop on Logic Synthesis 1991 (IWLS-91); Microelectronics Center of North Carolina, USA, May 1991.

[LGSynt93] Benchmarks set for the International Workshop on Logic Synthesis 1993 (IWLS-93); Tahoe City, USA, May 1993.

[Mermet93] Mermet, Jean P.; Fundamentals and Standards in Hardware Description Languages; Kluwer Academic Publishers, 1993.

[HsuTsa95] Hsu, Y., Tsai, K., Liu, J. und Lin, E.; VHDL Modeling for Digital Design Synthesis; Kluwer Academic Publishers, 1995.

[MicLau92] Michel, P., Lauther, U. und Duzy, P.; The Synthesis approach to digital system design; Kluwer Academic Publishers, 1992.

[MilCum99] Mills, D. und Cummings, C.; RTL Coding Styles That Yield Simulation and Synthesis Mismatches; Synopsys Users Group Workshop, San Jose, USA, March 1999.

[Renner81] Renner, Edmund; Mathematisch-statistische Methoden in der praktischen Anwendung; Pareys Studientexte, Band 31, ISBN 3-489-613, Verlag Paul Parey, 1981.

[Roggen99] Roggenbuck, M.; Modifikation der 68HC05 Quickcore CPU; Technische Universität Braunschweig, Abteilung E.I.S., Studienarbeit, 1999.

[ScWaTe94] Schäfers, M., Wachsmann, K.P., Telkamp, G.; The URISC Exercise; Fifth Eurochip Workshop on VLSI Design Training, Dresden, Germany, 1994.

[SeaBre92] Seawrigt, A., Brewer, F.; Synthesis from production-based specification; 29th Design Automation Conference, Anaheim, California, USA, 1992.

[SeaBre94] Seawrigt, A., Brewer, F.; Clairvoyant: A Synthesis System For Production-Based Specification; IEEE Transaction on VLSI Systems, volume 2, number 2, June 1994.

[SeaHol96] Seawright, A., Holtmann, U., Meyer, W., Pangrle, B., Verbrugghe, R. und Buck, J.; A System for Compiling an Debugging Structured Data Processing Controllers; European Design Automation Conference (EDAC), Geneva, Switzerland, 1996.

[Shepar97] Shepard, K. L.; Practical Issues of Interconnect Analysis in Deep Submicron Integrated Circuits; IEEE International Conference on Computer Design (ICCD), 1997.

[SonPhi83] Sony, Inc., Philips, Inc.; Digital audio interface for domestic use; 1983.

[SynBCM97] Synopsys; Behavior Compiler Methodology Guide, Version 1997.08; Synopsys Inc., 1997.

[SynBCU97] Synopsys; Behavior Compiler User Guide, Version 1997.08; Synopsys Inc., 1997.

[SynDCR97] Synopsys; Design Compiler Reference, Version 1997.08; Synopsys Inc., 1997.

[SynHDL97] Synopsys; HDL Compiler for Verilog Reference, Version 1997.08; Synopsys Inc., 1997.

[SynPCU99] Synopsys; Protocol Compiler User Guide, Version 1999.05; Synopsys Inc., 1999.

[VerCad97] Cadence; Verilog-XL Reference Manual, Version 2.5; Cadence Design Systems Inc., 1997.

[VerStd96] IEEE Computer Society; IEEE Std 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language; IEEE Computer Society, 1996.

[VerSyn99] IEEE Computer Society; IEEE P1364.1-1999, Standard for Verilog Register Transfer Level Synthesis; IEEE Computer Society, 1999.

[VHDL94] IEEE Computer Society; IEEE Std 1076-1993; IEEE Standard VHDL Language; IEEE Computer Society, 1994.

[WalSch91] Wall, L., Schwartz, R.; Programming perl; O‘Reilly & Associates, 1991.

[Xilinx96] Xilinx; The Programmable Logic Data Book; Xilinx Inc., 1996.

[Xilinx98] Xilinx; M1.5i Development System Reference Guide; Xilinx Inc., 1998.

Lebenslauf

Name:	Peter Blinzer
Titel:	Dipl.-Inform.
Geburtsdatum:	6. Januar 1969
Geburtsort:	Martin (Slowakei)
Staatsbürgerschaft:	deutsch

Ausbildungsweg:

09/1975 - 07/1976	Grundschule in Mutlangen
09/1976 - 07/1979	Grundschule in Beverstedt
09/1976 - 07/1981	Orientierungsstufe in Beverstedt
09/1981 - 05/1988	Kreisgymnasium Wesermünde in Bremerhaven
05/1988	Erlangung der allgemeinen Hochschulreife
07/1988 - 09/1989	Grundwehrdienst
10/1989 - 01/1995	Studium der Informatik an der TU Braunschweig
01/1995	Erlangung des Hochschulgrades “Diplom-Informatiker”
02/1995 - 01/2000	Wissenschaftlicher Mitarbeiter an der TU Braunschweig mit Promotionsarbeit im Bereich Entwurf integrierter Schaltungen

Anhang A

In diesem Anhang werden die Ergebnisdaten der im Hauptteil beschriebenen und ausgewerteten Syntheseläufe tabellarisch aufgelistet. Dabei wurden bei allen Experimenten die RTL-Syntheseoptionen für `set_flatten` nicht im Detail untersucht, da diese Optimierungen generell nicht vollständig durchführbar waren und die Ergebnisse verschlechterten. Für die Übersetzung mit dem Design-Compiler wurde hoher Optimierungsaufwand gewählt (`compile -map_effort high`).

A.1 Entwurf eines Bildschirmcontrollers

In diesem Abschnitt sind die Syntheseergebnisse für den Bildschirmcontroller discount entsprechend [Golze99] und [Friedr98] mit Variationen der Einstellungen für RTL-, High-Level- und Controllersynthese aufgeführt.

A.1.1 RTL-Synthese

Verwendet wurde das Vorlesungsbeispiel aus [Golze99] nach Anpassung an die RTL-Synthese und das RTL-Modell des Bildschirmcontrollers aus [Friedr98]. Die variierten Syntheseeinstellungen umfassen:

- Taktvorgabe
- Flächenbegrenzung (`set_max_area`)
- Strukturoptimierungen (`set_structure`, `boolean`, `timing`)
- Übersetzungsoptionen (`incremental_map` und `prioritize_min_paths`)

Die experimentell betrachteten Einstellungen werden im folgenden mit einer Kurzschreibweise erfaßt, welche die Wertkombinationen entsprechend Tabelle A.1 kodiert.

Einstellungscode	Wert	Bedeutung
T	0	keine Taktvorgabe
	1	Taktvorgabe 12,80MHz = Zielfrequenz -10%
	2	Taktvorgabe 14,32MHz = Zielfrequenz
	3	Taktvorgabe 15,70MHz = Zielfrequenz +10%
A	0	keine Flächenbegrenzung
	1	minimale Fläche (`set_max_area 0`)
S	0	keine Strukturierungsoption
	1	Strukturierungs-Optimierung mit Boolescher Option
	2	Strukturierungs-Optimierung mit Timing-Option
	3	Strukturierungs-Optimierung mit Boolescher und Timing-Option
C	0	`compile` ohne Zusatzoptionen
	1	`compile` mit `incremental_map`
	2	`compile` mit `prioritize_min_paths`
	3	`compile` mit `incremental_map` und `prioritize_min_paths`

Tabelle A.1: Kurzschreibweise der RTL-Syntheseeinstellungen für discount

Im Anschluß an die Synthese mit dem Design-Compiler 1997.08 [SynDCR97] wurde die Plazierung und Verdrahtung mit XACT 5.2.1 [Xilinx96] vorgenommen. Die Ergebnisse der variierten Syntheseläufe in Größe (CLBs) und FPGA-Taktrate (MHz) für den FPGA-Typ 3042-125 zu beiden Modellen zeigt Tabelle A.2.

Einstellung	RTL-Modell aus [Golze99]				RTL-Modell aus [Friedr98]			
	A0		A1		A0		A1	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
T0 S0 C0	066	13.00	066	12.47	053	12.13	053	11.93
T0 S0 C1	066	13.40	066	12.53	051	13.37	050	12.33
T0 S0 C2	066	13.30	066	12.97	053	12.37	053	12.17
T0 S0 C3	066	12.87	066	12.47	051	13.93	050	12.30
T0 S1 C0	057	15.63	057	15.60	052	12.07	052	11.47
T0 S1 C1	057	15.60	057	14.87	051	14.10	049	11.57
T0 S1 C2	057	15.50	057	15.70	052	11.37	052	12.30
T0 S1 C3	057	15.27	057	15.20	051	13.37	049	11.83
T0 S2 C0	055	15.27	055	15.20	053	12.70	053	11.77
T0 S2 C1	055	15.60	055	15.27	051	12.63	050	13.03
T0 S2 C2	055	15.13	055	16.00	053	11.57	053	12.43
T0 S2 C3	055	15.73	055	15.40	051	13.13	050	11.93
T0 S3 C0	057	14.90	057	14.73	052	11.83	052	11.87
T0 S3 C1	057	15.53	057	16.03	051	12.87	049	12.67
T0 S3 C2	057	15.07	057	15.60	052	12.07	052	11.70
T0 S3 C3	057	15.47	057	15.57	051	12.93	049	11.97
T1 S0 C0	065	15.40	065	14.77	051	12.60	051	13.03
T1 S0 C1	074	14.33	074	14.57	052	12.43	052	12.33
T1 S0 C2	065	14.70	065	14.47	051	13.50	051	12.37
T1 S0 C3	074	15.10	074	14.30	052	11.37	052	11.87
T1 S1 C0	056	14.87	056	14.47	051	14.17	051	13.97
T1 S1 C1	060	17.37	060	15.93	052	11.73	052	11.90
T1 S1 C2	056	14.87	056	15.13	051	13.80	051	13.83
T1 S1 C3	060	16.43	060	16.33	052	11.93	052	11.87
T1 S2 C0	057	13.73	057	14.40	051	13.87	051	13.73
T1 S2 C1	059	17.07	059	17.30	052	11.77	052	11.23
T1 S2 C2	057	13.77	057	14.47	051	13.17	051	13.40
T1 S2 C3	059	17.50	057	17.50	052	12.23	052	11.73
T1 S3 C0	056	15.00	056	14.93	051	13.93	051	13.93
T1 S3 C1	060	16.50	060	16.63	052	12.00	052	10.60
T1 S3 C2	056	15.13	056	15.27	051	13.43	051	13.77
T1 S3 C3	060	17.27	060	17.27	052	11.90	052	11.40
T2 S0 C0	065	15.53	065	14.20	052	13.10	052	12.37
T2 S0 C1	073	15.07	073	13.73	052	11.93	052	11.90
T2 S0 C2	065	14.27	065	14.20	052	12.70	052	13.27
T2 S0 C3	073	15.47	073	14.93	052	11.60	052	12.37
T2 S1 C0	056	14.60	056	15.00	051	13.87	051	13.60
T2 S1 C1	060	16.53	060	17.17	052	11.63	052	11.47
T2 S1 C2	056	15.20	056	14.43	051	13.43	051	14.17
T2 S1 C3	060	16.20	060	16.83	052	11.83	052	11.97
T2 S2 C0	057	13.80	057	14.97	051	13.67	051	13.53
T2 S2 C1	060	16.43	060	16.83	052	11.87	052	11.93
T2 S2 C2	057	14.63	057	14.77	051	12.97	051	13.10
T2 S2 C3	060	16.70	060	15.67	052	12.10	052	11.63
T2 S3 C0	056	15.47	056	14.43	051	12.90	051	13.50
T2 S3 C1	060	16.73	060	16.80	052	11.33	052	11.97
T2 S3 C2	056	14.43	056	14.50	051	13.70	051	13.93
T2 S3 C3	060	16.73	060	16.43	052	12.07	052	11.80
T3 S0 C0	065	13.77	065	14.33	051	10.93	051	10.87

Tabelle A.2: Ergebnisse der RTL-Synthese für discount

Einstellung	RTL-Modell aus [Golze99]				RTL-Modell aus [Friedr98]			
	A0		A1		A0		A1	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
T3 S0 C1	073	14.50	073	14.77	053	11.23	053	11.50
T3 S0 C2	065	15.03	065	15.57	051	11.33	051	11.10
T3 S0 C3	073	14.67	073	14.40	053	12.00	053	12.57
T3 S1 C0	056	14.33	056	15.37	051	14.93	051	13.97
T3 S1 C1	060	16.87	060	16.50	053	10.80	053	11.97
T3 S1 C2	056	15.20	056	14.50	051	14.00	051	14.13
T3 S1 C3	060	16.33	060	16.43	053	12.37	053	11.80
T3 S2 C0	057	14.47	057	14.33	051	13.70	051	13.93
T3 S2 C1	060	16.63	060	17.10	053	11.73	053	11.57
T3 S2 C2	057	14.93	057	14.00	051	13.77	051	13.27
T3 S2 C3	060	17.63	060	14.80	053	11.90	053	11.27
T3 S3 C0	056	14.80	056	15.00	051	13.40	051	13.17
T3 S3 C1	060	16.70	060	16.23	053	11.57	053	11.87
T3 S3 C2	056	14.63	056	14.83	051	13.37	051	14.10
T3 S3 C3	060	16.43	060	16.67	053	11.37	053	11.27

Tabelle A.2: Ergebnisse der RTL-Synthese für discount

A.1.2 High-Level-Synthese

Es wurde das Modell des Bildschirmcontrollers aus [Friedr98] untersucht, bei dem die High-Level-Syntheseoptionen fest gewählt wurden. Kleinere Änderungen in der Taktspezifikation oder den Zielbibliotheken hatten stark negativen Einfluß auf die Syntheseergebnisse, bis hin zur Undurchführbarkeit des Syntheselaufs. Die variierten RTL-Syntheseeinstellungen umfassen:

- Flächenbegrenzung (`set_max_area`)
- Strukturoptimierungen (`set_structure`, `boolean`, `timing`)
- Übersetzungsoptionen (`incremental_map` und `prioritize_min_paths`)

Nach der High-Level- und RTL-Synthese mit dem Behavior-Compiler [SynBCU97] bzw. dem Design-Compiler 1997.08 wurde XACT 5.2.1 zur FPGA-Abbildung benutzt. Die Ergebnisse der Variationen zeigt Tabelle A.3, wobei die Kurzbezeichnungen aus Tabelle A.1 verwendet werden.

Optionen	High-Level-Modell aus [Friedr98] auf FPGA 3164A-2							
	C0		C1		C2		C3	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
A0 S0	112	15.23	182	11.23	112	14.87	182	10.83
A0 S1	110	13.50	170	9.60	110	13.57	170	9.07
A0 S2	097	13.57	170	9.50	097	13.83	170	10.03
A0 S3	110	14.23	170	9.73	110	13.57	170	9.97
A1 S0	112	14.60	182	10.80	112	14.37	182	10.53
A1 S1	110	13.87	170	9.30	110	13.33	170	10.07
A1 S2	097	14.13	170	9.47	097	13.57	170	8.87
A1 S3	110	13.83	170	8.77	110	13.43	170	9.50

Tabelle A.3: Ergebnisse der High-Level-Synthese für discount

A.1.3 Controllersynthese

Als Basis für diese Synthesevariationen diente eine hierarchiefreie Beschreibung des Bildschirmcontrollers aus verschachtelten Wiederholungen, die über Zähler gesteuert werden (Bild 6.9). Die Controllersynthese mit dem Protocol-Compiler 1999.10-beta2 erzeugte verschiedene RTL-Beschreibungen durch:

- drei mögliche HDL-Modellstrukturen
- sechs Arten der Zustandscodierung
- einzelne LFSR-Zähler oder gemeinsam genutzte Binärzähler

Bei der RTL-Synthese mit dem Design-Compiler 1997.08 wurden Taktvorgabe, Flächenbegrenzung, Strukturoptimierungen und `compile`-Optionen verändert (Tabelle A.1). Die Plazierung und Verdrahtung des FPGAs 3090A-7 erfolgte mit Xilinx-M1.5i [Xilinx98]. Die insgesamt 2304 Ergebnistupel für Schaltungsgröße (CLBs) und Geschwindigkeit (MHz) zeigt Tabelle A.4.

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
Single Process Automatic Binärzähler	S0 C0	107	10.64	103	9.46	103	13.08	103	13.08
	S0 C1	106	10.03	103	9.43	119	7.42	119	7.42
	S0 C2	107	10.64	103	9.46	103	13.08	103	13.08
	S0 C3	106	10.03	103	9.43	119	7.42	119	7.42
	S1 C0	072	11.60	072	9.81	072	11.50	072	11.50
	S1 C1	073	11.79	071	10.93	085	9.60	085	9.60
	S1 C2	072	11.60	072	9.81	072	11.50	072	11.50
	S1 C3	073	11.79	071	10.93	085	9.60	085	9.60
	S2 C0	081	11.46	081	10.00	073	13.48	073	13.48
	S2 C1	081	8.04	081	11.33	087	9.98	087	9.98
	S2 C2	081	11.46	081	10.00	073	13.48	073	13.48
	S2 C3	081	8.04	081	11.33	087	9.98	087	9.98
	S3 C0	072	11.60	072	9.81	072	11.50	072	11.50
	S3 C1	073	11.79	071	10.93	084	9.80	084	9.80
	S3 C2	072	11.60	072	9.81	072	11.50	072	11.50
	S3 C3	073	11.79	071	10.93	084	9.80	084	9.80
Single Process Automatic LFSR-Zähler	S0 C0	100	10.24	100	10.65	103	13.87	103	13.87
	S0 C1	101	11.90	102	11.14	109	11.36	109	11.36
	S0 C2	100	10.24	100	10.65	103	13.87	103	13.87
	S0 C3	101	11.90	102	11.14	109	11.36	109	11.36
	S1 C0	085	7.92	084	7.87	087	11.81	087	11.81
	S1 C1	086	8.80	085	8.83	094	9.61	094	9.61
	S1 C2	085	7.92	084	7.87	087	11.81	087	11.81
	S1 C3	086	8.80	085	8.83	094	9.61	094	9.61
	S2 C0	087	11.07	089	11.52	090	14.36	090	14.36
	S2 C1	088	11.22	087	10.70	092	12.03	092	12.03
	S2 C2	087	11.07	089	11.52	090	14.36	090	14.36
	S2 C3	088	11.22	087	10.70	092	12.03	092	12.03
	S3 C0	085	7.92	084	7.87	087	11.81	087	11.81
	S3 C1	086	8.80	085	8.83	100	10.37	100	10.37
	S3 C2	085	7.92	084	7.87	087	11.81	087	11.81
	S3 C3	086	8.80	085	8.83	100	10.37	100	10.37

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
Single Process	S0 C0	166	4.60	165	4.69	147	6.14	147	6.14
Distributed	S0 C1	167	4.44	167	4.03	155	5.88	155	5.88
Binärzähler	S0 C2	166	4.60	165	4.69	147	6.14	147	6.14
	S0 C3	167	4.44	167	4.03	155	5.88	155	5.88
	S1 C0	114	4.72	114	5.59	097	6.43	097	6.43
	S1 C1	116	5.15	116	5.61	109	7.06	109	7.06
	S1 C2	114	4.72	114	5.59	097	6.43	097	6.43
	S1 C3	116	5.15	116	5.61	109	7.06	109	7.06
	S2 C0	119	5.55	118	5.62	098	7.87	098	7.87
	S2 C1	119	5.41	118	4.88	110	6.60	110	6.60
	S2 C2	119	5.55	118	5.62	098	7.87	098	7.87
	S2 C3	119	5.41	118	4.88	110	6.60	110	6.60
	S3 C0	114	4.72	114	5.59	096	6.13	096	6.13
	S3 C1	116	5.15	116	5.61	109	7.06	109	7.06
	S3 C2	114	4.72	114	5.59	096	6.13	096	6.13
	S3 C3	116	5.15	116	5.61	109	7.06	109	7.06
Single Process	S0 C0	145	5.71	145	6.37	133	8.36	133	8.36
Distributed	S0 C1	145	5.06	145	6.51	152	6.49	152	6.49
LFSR-Zähler	S0 C2	145	5.71	145	6.37	133	8.36	133	8.36
	S0 C3	145	5.06	145	6.51	152	6.49	152	6.49
	S1 C0	125	6.33	125	6.23	110	7.67	110	7.67
	S1 C1	126	6.19	125	6.67	119	7.78	119	7.78
	S1 C2	125	6.33	125	6.23	110	7.67	110	7.67
	S1 C3	126	6.19	125	6.67	119	7.78	119	7.78
	S2 C0	124	6.21	123	6.92	116	8.62	116	8.62
	S2 C1	124	7.06	123	6.62	126	6.48	126	6.48
	S2 C2	124	6.21	123	6.92	116	8.62	116	8.62
	S2 C3	124	7.06	123	6.62	126	6.48	126	6.48
	S3 C0	125	6.33	125	6.23	110	7.67	110	7.67
	S3 C1	126	6.19	125	6.67	124	6.59	124	6.59
	S3 C2	125	6.33	125	6.23	110	7.67	110	7.67
	S3 C3	126	6.19	125	6.67	124	6.59	124	6.59
Single Process	S0 C0	178	4.95	177	4.38	164	7.02	164	7.02
Binary-Min	S0 C1	176	4.14	177	4.32	199	4.51	199	4.51
Binärzähler	S0 C2	178	4.95	177	4.38	164	7.02	164	7.02
	S0 C3	176	4.14	177	4.32	199	4.51	199	4.51
	S1 C0	117	3.97	118	3.50	096	5.33	096	5.33
	S1 C1	117	3.87	118	3.72	119	3.88	119	3.88
	S1 C2	117	3.97	118	3.50	096	5.33	096	5.33
	S1 C3	117	3.87	118	3.72	119	3.88	119	3.88
	S2 C0	149	5.49	147	5.04	121	6.83	121	6.83
	S2 C1	149	5.32	147	4.41	152	5.02	152	5.02
	S2 C2	149	5.49	147	5.04	121	6.83	121	6.83
	S2 C3	149	5.32	147	4.41	152	5.02	152	5.02
	S3 C0	117	3.98	118	4.21	089	5.33	089	5.33
	S3 C1	117	4.01	118	4.47	122	4.37	122	4.37
	S3 C2	117	3.98	118	4.21	089	5.33	089	5.33
	S3 C3	117	4.01	118	4.47	122	4.37	122	4.37
Single Process	S0 C0	138	7.02	138	5.99	127	7.72	127	7.72
Binary-Min	S0 C1	137	6.31	138	6.09	176	4.99	176	4.99
LFSR-Zähler	S0 C2	138	7.02	138	5.99	127	7.72	127	7.72

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S0 C3	137	6.31	138	6.09	176	4.99	176	4.99
	S1 C0	117	4.41	117	4.85	101	3.92	101	3.92
	S1 C1	117	4.76	117	4.60	133	4.15	133	4.15
	S1 C2	117	4.41	117	4.85	101	3.92	101	3.92
	S1 C3	117	4.76	117	4.60	133	4.15	133	4.15
	S2 C0	139	6.23	139	6.64	116	7.34	116	7.34
	S2 C1	133	5.83	131	7.20	150	5.08	150	5.08
	S2 C2	139	6.23	139	6.64	116	7.34	116	7.34
	S2 C3	133	5.83	131	7.20	150	5.08	150	5.08
	S3 C0	117	4.41	117	4.85	103	4.83	103	4.83
	S3 C1	117	4.76	117	4.60	135	4.05	103	4.83
	S3 C2	117	4.41	117	4.85	103	4.83	103	4.83
	S3 C3	117	4.76	117	4.60	135	4.05	135	4.05
Single Process Min-Distance Binärzähler	S0 C0	179	3.69	178	3.96	165	7.80	165	7.80
	S0 C1	180	4.39	180	4.50	198	3.84	198	3.84
	S0 C2	179	3.69	178	3.96	165	7.80	165	7.80
	S0 C3	180	4.39	180	4.50	198	3.84	198	3.84
	S1 C0	128	2.83	128	2.54	105	3.34	105	3.34
	S1 C1	127	2.53	127	2.45	119	3.71	119	3.71
	S1 C2	128	2.83	128	2.54	105	3.34	105	3.34
	S1 C3	127	2.53	127	2.45	119	3.71	119	3.71
	S2 C0	159	4.94	158	4.28	139	6.52	139	6.52
	S2 C1	159	4.27	157	4.79	156	5.32	156	5.32
	S2 C2	159	4.94	158	4.28	139	6.52	139	6.52
	S2 C3	159	4.27	157	4.79	156	5.32	156	5.32
	S3 C0	127	2.72	127	2.76	105	3.99	105	3.99
	S3 C1	127	2.39	127	2.67	122	3.12	122	3.12
	S3 C2	127	2.72	127	2.76	105	3.99	105	3.99
	S3 C3	127	2.39	127	2.67	122	3.12	122	3.12
Single Process Min-Distance LFSR-Zähler	S0 C0	141	5.62	140	6.11	122	3.65	130	9.49
	S0 C1	140	5.18	140	5.52	176	5.32	176	5.32
	S0 C2	141	5.62	140	6.11	130	9.49	130	9.49
	S0 C3	140	5.18	140	5.52	176	5.32	176	5.32
	S1 C0	122	3.71	124	3.93	114	4.11	114	4.11
	S1 C1	121	4.24	122	3.65	129	3.22	129	3.22
	S1 C2	122	3.71	124	3.93	114	4.11	114	4.11
	S1 C3	121	4.24	122	3.65	129	3.22	129	3.22
	S2 C0	130	6.58	129	6.76	117	7.44	117	7.44
	S2 C1	130	6.62	129	6.32	140	6.10	140	6.10
	S2 C2	130	6.58	129	6.76	117	7.44	117	7.44
	S2 C3	130	6.62	129	6.32	140	6.10	140	6.10
	S3 C0	122	3.71	124	3.93	103	4.69	103	4.69
	S3 C1	121	4.24	122	3.65	128	3.50	128	3.50
	S3 C2	122	3.71	124	3.93	103	4.69	103	4.69
	S3 C3	121	4.24	122	3.65	128	3.50	128	3.50
Single Process Branch-Sequence Binärzähler	S0 C0	173	3.66	171	4.32	151	7.52	151	7.52
	S0 C1	171	4.07	171	4.75	183	4.92	183	4.92
	S0 C2	173	3.66	171	4.32	151	7.52	151	7.52
	S0 C3	171	4.07	171	4.75	183	4.92	183	4.92
	S1 C0	107	3.15	107	3.03	079	4.90	079	4.90
	S1 C1	107	3.27	107	3.03	108	4.89	108	4.89

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S1 C2	107	3.15	107	3.03	079	4.90	079	4.90
	S1 C3	107	3.27	107	3.03	108	4.89	108	4.89
	S2 C0	135	3.92	135	4.77	109	8.15	109	8.15
	S2 C1	135	4.36	135	4.79	138	6.37	138	6.37
	S2 C2	135	3.92	135	4.77	138	5.39	109	8.15
	S2 C3	135	4.36	135	4.79	138	6.37	138	6.37
	S3 C0	107	3.15	107	3.03	074	4.38	074	4.38
	S3 C1	107	3.27	107	3.03	108	4.89	108	4.89
	S3 C2	107	3.15	107	3.03	074	4.38	074	4.38
	S3 C3	107	3.27	107	3.03	108	4.89	108	4.89
Single Process Branch-Sequence LFSR-Zähler	S0 C0	134	6.65	133	6.00	131	10.82	131	10.82
	S0 C1	132	5.78	134	6.19	171	4.96	171	4.96
	S0 C2	134	6.65	133	6.00	131	10.82	131	10.82
	S0 C3	132	5.78	134	6.19	171	4.96	171	4.96
	S1 C0	112	4.26	112	4.48	098	4.02	098	4.02
	S1 C1	112	4.12	113	4.38	135	4.40	135	4.40
	S1 C2	112	4.26	112	4.48	098	4.02	098	4.02
	S1 C3	112	4.12	113	4.38	135	4.40	135	4.40
	S2 C0	126	5.91	124	6.50	114	8.50	114	8.50
	S2 C1	126	6.46	124	7.12	143	5.81	143	5.81
	S2 C2	126	5.91	124	6.50	114	8.50	114	8.50
	S2 C3	126	6.46	124	7.12	143	5.81	143	5.81
	S3 C0	112	4.26	112	4.48	098	3.66	098	3.66
	S3 C1	112	4.12	113	4.38	135	4.40	135	4.40
	S3 C2	112	4.26	112	4.48	098	3.66	098	3.66
	S3 C3	112	4.12	113	4.38	135	4.40	135	4.40
Single Process One-Hot Binärzähler	S0 C0	177	4.28	175	4.98	161	6.62	161	6.62
	S0 C1	175	4.49	173	3.77	184	5.65	184	5.65
	S0 C2	177	4.28	175	4.98	161	6.62	161	6.62
	S0 C3	175	4.49	173	3.77	184	5.65	184	5.65
	S1 C0	124	6.20	124	5.49	102	6.33	102	6.33
	S1 C1	125	5.50	124	5.63	115	6.99	115	6.99
	S1 C2	124	6.20	124	5.49	102	6.33	102	6.33
	S1 C3	125	5.50	124	5.63	115	6.99	115	6.99
	S2 C0	124	5.20	125	6.03	104	9.63	104	9.63
	S2 C1	125	5.26	125	5.16	118	6.84	118	6.84
	S2 C2	124	5.20	125	6.03	104	9.63	104	9.63
	S2 C3	125	5.26	125	5.16	118	6.84	118	6.84
	S3 C0	124	6.20	124	5.49	102	6.33	102	6.33
	S3 C1	125	5.50	124	5.63	114	8.37	114	8.37
	S3 C2	124	6.20	124	5.49	102	6.33	102	6.33
	S3 C3	125	5.50	124	5.63	114	8.37	114	8.37
Single Process One-Hot LFSR-Zähler	S0 C0	164	6.77	162	7.01	156	7.79	156	7.79
	S0 C1	163	6.17	162	6.88	168	6.58	168	6.58
	S0 C2	164	6.77	162	7.01	156	7.79	156	7.79
	S0 C3	163	6.17	162	6.88	168	6.58	168	6.58
	S1 C0	129	5.67	128	6.94	119	7.59	119	7.59
	S1 C1	129	6.54	128	6.46	128	7.23	128	7.23
	S1 C2	129	5.67	128	6.94	119	7.59	119	7.59
	S1 C3	129	6.54	128	6.46	128	7.23	128	7.23
	S2 C0	134	6.96	134	7.10	128	8.11	128	8.11

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S2 C1	134	6.12	133	7.17	136	8.49	136	8.49
	S2 C2	134	6.96	134	7.10	128	8.11	128	8.11
	S2 C3	134	6.12	133	7.17	136	8.49	136	8.49
	S3 C0	129	5.67	128	6.94	125	8.48	125	8.48
	S3 C1	129	6.54	128	6.46	126	6.88	126	6.88
	S3 C2	129	5.67	128	6.94	125	8.48	125	8.48
	S3 C3	129	6.54	128	6.46	126	6.88	126	6.88
Split Process	S0 C0	126	10.04	126	8.89	126	9.72	126	9.72
Automatic	S0 C1	124	9.18	124	11.37	137	9.53	137	9.53
Binärzähler	S0 C2	126	10.04	126	8.89	126	9.72	126	9.72
	S0 C3	124	9.18	124	11.37	126	11.02	137	9.53
	S1 C0	090	9.42	090	10.65	083	11.65	083	11.65
	S1 C1	091	9.17	090	9.57	093	10.75	093	10.75
	S1 C2	090	9.42	090	10.65	083	11.65	083	11.65
	S1 C3	091	9.17	090	9.57	093	10.75	093	10.75
	S2 C0	094	11.21	095	11.94	085	12.25	085	12.25
	S2 C1	093	10.34	094	11.23	116	8.64	116	8.64
	S2 C2	094	11.21	095	11.94	085	12.25	085	12.25
	S2 C3	093	10.34	094	11.23	116	8.64	116	8.64
	S3 C0	090	9.42	090	10.65	090	10.37	090	10.37
	S3 C1	091	9.17	090	9.57	100	9.57	100	9.57
	S3 C2	090	9.42	090	10.65	090	10.37	090	10.37
	S3 C3	091	9.17	090	9.57	100	9.57	100	9.57
Split Process	S0 C0	118	9.49	120	9.72	124	14.54	124	14.54
Automatic	S0 C1	119	10.02	120	10.76	129	8.70	129	8.70
LFSR-Zähler	S0 C2	118	9.49	120	9.72	124	14.54	124	14.54
	S0 C3	119	10.02	120	10.76	129	8.70	129	8.70
	S1 C0	101	11.40	099	10.21	101	13.84	101	13.84
	S1 C1	101	10.20	100	10.21	112	11.29	112	11.29
	S1 C2	101	11.40	099	10.21	101	13.84	101	13.84
	S1 C3	101	10.20	100	10.21	112	11.29	112	11.29
	S2 C0	104	11.03	103	13.14	099	14.14	099	14.14
	S2 C1	105	11.95	105	10.91	124	9.52	124	9.52
	S2 C2	104	11.03	103	13.14	124	10.14	099	14.14
	S2 C3	105	11.95	105	10.91	124	9.52	124	9.52
	S3 C0	101	11.40	099	10.21	101	11.49	101	11.49
	S3 C1	101	10.20	100	10.21	122	9.83	122	9.83
	S3 C2	101	11.40	099	10.21	101	11.49	101	11.49
	S3 C3	101	10.20	100	10.21	122	9.83	122	9.83
Split Process	S0 C0	221	4.25	221	4.45	196	5.57	196	5.57
Distributed	S0 C1	222	3.86	224	4.25	210	4.70	210	4.70
Binärzähler	S0 C2	221	4.25	221	4.45	196	5.57	196	5.57
	S0 C3	222	3.86	224	4.25	210	4.70	210	4.70
	S1 C0	131	5.10	130	5.15	116	4.56	116	4.56
	S1 C1	131	5.40	130	4.58	124	4.68	124	4.68
	S1 C2	131	5.10	130	5.15	116	4.56	116	4.56
	S1 C3	131	5.40	130	4.58	124	4.68	124	4.68
	S2 C0	136	4.84	135	4.50	119	6.73	119	6.73
	S2 C1	135	5.03	135	4.44	140	6.29	140	6.29
	S2 C2	136	4.84	135	4.50	119	6.73	119	6.73
	S2 C3	135	5.03	135	4.44	140	6.29	140	6.29

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S3 C0	131	5.10	130	5.15	113	5.66	113	5.66
	S3 C1	131	5.40	130	4.58	123	5.74	123	5.74
	S3 C2	131	5.10	130	5.15	113	6.15	113	6.15
	S3 C3	131	5.40	130	4.58	123	5.74	123	5.74
Split Process	S0 C0	213	5.72	213	5.63	202	5.93	202	5.93
Distributed	S0 C1	211	5.61	211	5.92	228	4.70	228	4.70
LFSR-Zähler	S0 C2	213	5.72	213	5.63	202	5.93	202	5.93
	S0 C3	211	5.61	211	5.92	228	4.70	228	4.70
	S1 C0	156	3.84	155	4.48	138	5.73	138	5.73
	S1 C1	154	3.88	154	3.27	156	5.17	156	5.17
	S1 C2	156	3.84	155	4.48	138	5.73	138	5.73
	S1 C3	154	3.88	154	3.27	156	5.17	156	5.17
	S2 C0	159	6.51	158	7.24	152	6.53	152	6.53
	S2 C1	160	6.27	158	6.49	184	4.28	184	4.28
	S2 C2	159	6.51	158	7.24	152	6.53	152	6.53
	S2 C3	160	6.27	158	6.49	184	4.28	184	4.28
	S3 C0	156	3.84	155	4.48	134	6.07	134	6.07
	S3 C1	154	3.88	154	3.27	173	5.58	173	5.58
	S3 C2	156	3.84	155	4.48	134	6.07	134	6.07
	S3 C3	154	3.88	154	3.27	173	5.58	173	5.58
Split Process	S0 C0	234	4.49	234	4.47	212	4.94	212	4.94
Binary-Min	S0 C1	237	3.83	234	4.47	258	4.01	258	4.01
Binärzähler	S0 C2	237	3.83	234	4.47	212	4.94	212	4.94
	S0 C3	237	3.83	236	3.39	258	4.01	258	4.01
	S1 C0	127	3.11	128	2.89	111	3.12	111	3.12
	S1 C1	126	2.81	127	3.27	149	2.97	149	2.97
	S1 C2	127	3.11	128	2.89	111	3.12	111	3.12
	S1 C3	126	2.81	127	3.27	149	2.97	149	2.97
	S2 C0	156	5.08	157	4.63	139	5.89	139	5.89
	S2 C1	155	4.29	157	4.60	184	4.39	184	4.39
	S2 C2	156	5.08	157	4.63	139	5.89	139	5.89
	S2 C3	155	4.29	157	4.60	184	4.39	184	4.39
	S3 C0	127	3.11	128	2.89	104	3.94	104	3.94
	S3 C1	126	2.81	127	3.27	149	2.45	149	2.45
	S3 C2	127	3.11	128	2.89	104	3.94	104	3.94
	S3 C3	126	2.81	127	3.27	149	2.45	149	2.45
Split Process	S0 C0	187	5.23	188	4.70	185	8.52	185	8.52
Binary-Min	S0 C1	190	5.59	188	6.25	250	4.43	250	4.43
LFSR-Zähler	S0 C2	187	5.23	188	4.70	185	8.52	185	8.52
	S0 C3	190	5.59	188	6.25	250	4.43	250	4.43
	S1 C0	143	2.76	142	2.70	141	3.10	141	3.10
	S1 C1	143	2.60	144	2.65	168	2.88	168	2.88
	S1 C2	143	2.76	142	2.70	141	3.10	141	3.10
	S1 C3	143	2.60	144	2.65	168	2.88	168	2.88
	S2 C0	154	5.47	156	6.06	147	8.84	147	8.84
	S2 C1	155	6.12	156	5.65	207	5.13	207	5.13
	S2 C2	154	5.47	156	6.06	147	8.84	147	8.84
	S2 C3	155	6.12	156	5.65	207	5.13	207	5.13
	S3 C0	143	2.76	142	2.70	145	4.56	145	4.56
	S3 C1	143	2.60	144	2.65	180	3.44	180	3.44
	S3 C2	143	2.76	142	2.70	145	4.56	145	4.56

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S3 C3	143	2.60	144	2.65	180	3.44	180	3.44
Split Process	S0 C0	240	4.29	240	3.71	217	5.24	217	5.24
Min-Distance	S0 C1	240	4.29	238	4.80	265	3.97	265	3.97
Binärzähler	S0 C2	240	4.29	240	3.71	217	5.24	217	5.24
	S0 C3	240	4.29	238	4.80	265	3.97	265	3.97
	S1 C0	132	2.94	132	2.83	123	3.69	123	3.69
	S1 C1	132	2.49	132	2.25	155	2.66	155	2.66
	S1 C2	132	2.94	132	2.83	123	3.69	123	3.69
	S1 C3	132	2.49	132	2.25	155	2.66	155	2.66
	S2 C0	161	4.73	163	5.03	150	5.72	150	5.72
	S2 C1	161	3.75	163	4.52	227	4.67	227	4.67
	S2 C2	161	4.73	163	5.03	150	5.72	150	5.72
	S2 C3	161	3.75	163	4.52	227	4.67	227	4.67
	S3 C0	132	2.94	132	2.83	120	3.68	120	3.68
	S3 C1	132	2.49	132	2.25	156	2.86	156	2.86
	S3 C2	132	2.94	132	2.83	120	3.68	120	3.68
	S3 C3	132	2.49	132	2.25	156	2.86	156	2.86
Split Process	S0 C0	194	5.89	194	6.39	194	7.10	194	7.10
Min-Distance	S0 C1	196	4.99	195	5.01	249	5.12	249	5.12
LFSR-Zähler	S0 C2	194	5.89	194	6.39	194	7.10	194	7.10
	S0 C3	196	4.99	195	5.01	249	5.12	249	5.12
	S1 C0	149	2.84	151	2.72	138	4.56	138	4.56
	S1 C1	146	2.80	145	2.43	170	3.28	170	3.28
	S1 C2	149	2.84	151	2.72	138	4.56	138	4.56
	S1 C3	146	2.80	145	2.43	170	3.28	170	3.28
	S2 C0	158	5.34	157	5.48	149	9.07	149	9.07
	S2 C1	158	5.89	157	5.82	230	4.17	230	4.17
	S2 C2	158	5.34	157	5.48	149	9.07	149	9.07
	S2 C3	158	5.89	157	5.82	230	4.17	230	4.17
	S3 C0	149	2.84	151	2.72	140	4.50	140	4.50
	S3 C1	146	2.80	145	2.43	178	3.78	178	3.78
	S3 C2	149	2.84	151	2.72	140	4.50	140	4.50
	S3 C3	146	2.80	145	2.43	178	3.78	178	3.78
Split Process	S0 C0	227	4.78	228	5.42	200	6.77	200	6.77
Branch-Sequence	S0 C1	228	5.13	229	4.28	251	4.07	251	4.07
Binärzähler	S0 C2	227	4.78	228	5.42	200	6.77	200	6.77
	S0 C3	228	5.13	229	4.28	251	4.07	251	4.07
	S1 C0	115	3.50	116	3.58	101	4.38	101	4.38
	S1 C1	115	3.75	116	3.03	135	3.60	135	3.60
	S1 C2	115	3.50	116	3.58	101	4.38	101	4.38
	S1 C3	115	3.75	116	3.03	135	3.60	135	3.60
	S2 C0	145	3.92	145	4.33	121	7.46	121	7.46
	S2 C1	145	4.30	145	4.73	172	4.07	172	4.07
	S2 C2	145	3.92	145	4.33	121	7.46	121	7.46
	S2 C3	145	4.30	145	4.73	172	4.07	172	4.07
	S3 C0	115	3.50	116	3.58	101	4.14	101	4.14
	S3 C1	115	3.75	116	3.03	132	4.06	132	4.06
	S3 C2	115	3.50	116	3.58	101	4.14	101	4.14
	S3 C3	115	3.75	116	3.03	132	4.06	132	4.06
Split Process	S0 C0	185	5.75	185	5.50	186	5.83	186	5.83
Branch-Sequence	S0 C1	187	5.98	185	5.77	246	4.76	246	4.76

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
LFSR-Zähler	S0 C2	185	5.75	185	5.50	186	5.83	186	5.83
	S0 C3	187	5.98	185	5.50	246	4.76	246	4.76
	S1 C0	146	3.24	146	3.45	130	3.05	130	3.05
	S1 C1	146	2.45	147	2.63	169	3.71	169	3.71
	S1 C2	146	3.24	146	3.45	130	3.05	130	3.05
	S1 C3	146	2.45	147	2.63	169	3.71	169	3.71
	S2 C0	150	5.64	150	5.68	148	8.28	148	8.28
	S2 C1	151	5.48	149	6.07	191	4.74	191	4.74
	S2 C2	150	5.64	150	5.68	148	8.28	148	8.28
	S2 C3	150	5.64	149	6.07	191	4.74	191	4.74
	S3 C0	146	3.24	146	3.45	124	3.15	124	3.15
	S3 C1	146	2.45	147	2.63	162	3.45	162	3.45
	S3 C2	146	3.24	146	3.45	124	3.15	124	3.15
	S3 C3	146	2.45	147	2.63	162	3.45	162	3.45
Split Process One-Hot Binärzähler	S0 C0	232	3.63	231	3.89	214	5.00	214	5.00
	S0 C1	232	3.67	231	5.14	242	5.49	242	5.49
	S0 C2	232	3.63	231	3.89	214	5.00	214	5.00
	S0 C3	232	3.67	231	5.14	242	5.49	242	5.49
	S1 C0	137	3.92	136	3.90	136	4.03	136	4.03
	S1 C1	135	3.48	135	3.38	140	4.84	140	4.84
	S1 C2	137	3.92	136	3.90	136	4.03	136	4.03
	S1 C3	135	3.48	135	3.38	140	4.84	140	4.84
	S2 C0	140	5.48	140	4.86	134	8.53	134	8.53
	S2 C1	139	5.00	141	4.38	191	5.42	191	5.42
	S2 C2	140	5.48	140	4.86	133	8.30	133	8.30
	S2 C3	139	5.00	141	4.38	191	5.42	191	5.42
	S3 C0	137	3.92	136	3.90	129	5.67	129	5.67
	S3 C1	135	3.48	136	3.90	145	5.43	145	5.43
	S3 C2	137	3.92	136	3.90	129	5.67	129	5.67
	S3 C3	135	3.48	135	3.38	145	5.43	145	5.43
Split Process One-Hot LFSR-Zähler	S0 C0	220	5.36	221	6.64	230	4.89	230	4.89
	S0 C1	225	5.42	226	6.59	249	5.33	249	5.33
	S0 C2	220	5.36	221	6.64	230	4.89	230	4.89
	S0 C3	225	5.42	226	6.59	249	5.33	249	5.33
	S1 C0	158	4.24	159	4.38	154	5.01	154	5.01
	S1 C1	154	4.38	155	3.99	165	4.79	165	4.79
	S1 C2	158	4.24	159	4.38	154	5.01	154	5.01
	S1 C3	154	4.38	155	3.99	165	4.79	165	4.79
	S2 C0	169	6.33	167	6.81	155	7.15	155	7.15
	S2 C1	168	6.25	167	6.55	205	4.43	205	4.43
	S2 C2	169	6.33	167	6.81	155	7.15	155	7.15
	S2 C3	168	6.25	167	6.55	205	4.43	205	4.43
	S3 C0	158	4.24	159	4.38	161	6.01	161	6.01
	S3 C1	154	4.38	155	3.99	178	4.25	178	4.25
	S3 C2	158	4.24	159	4.38	161	6.01	161	6.01
	S3 C3	154	4.38	155	3.99	178	4.25	178	4.25
Multi Process Automatic Binärzähler	S0 C0	125	10.29	125	10.69	125	12.63	125	12.63
	S0 C1	123	9.59	123	10.37	136	8.72	136	8.72
	S0 C2	125	10.29	125	10.69	125	12.63	125	12.63
	S0 C3	123	9.59	123	10.37	136	8.72	136	8.72
	S1 C0	090	10.57	090	10.57	083	12.09	083	12.09

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S1 C1	091	10.38	090	10.13	089	10.54	089	10.54
	S1 C2	090	10.57	090	10.57	083	12.09	083	12.09
	S1 C3	091	10.38	090	10.13	089	10.54	089	10.54
	S2 C0	094	11.54	095	12.34	085	12.42	085	12.42
	S2 C1	093	10.85	094	11.65	118	9.24	118	9.24
	S2 C2	094	11.54	095	12.34	085	12.42	085	12.42
	S2 C3	093	10.85	094	11.65	118	9.24	118	9.24
	S3 C0	090	10.57	090	10.57	090	10.39	090	10.39
	S3 C1	091	10.38	090	10.13	099	8.10	099	8.10
	S3 C2	090	10.57	090	10.57	090	10.39	090	10.39
	S3 C3	091	10.38	090	10.13	099	8.10	099	8.10
Multi Process Automatic LFSR-Zähler	S0 C0	117	9.83	118	10.73	124	13.76	124	13.76
	S0 C1	120	9.95	120	9.95	128	9.04	128	9.04
	S0 C2	117	9.83	118	10.73	124	13.76	124	13.76
	S0 C3	120	9.95	120	9.95	128	9.04	128	9.04
	S1 C0	101	10.81	099	10.13	103	14.61	103	14.61
	S1 C1	101	10.43	099	10.09	109	10.73	109	10.73
	S1 C2	101	10.81	099	10.13	103	14.61	103	14.61
	S1 C3	101	10.43	099	10.09	109	10.73	109	10.73
	S2 C0	103	11.27	103	11.31	098	14.54	098	14.54
	S2 C1	103	11.79	102	10.65	123	10.66	123	10.66
	S2 C2	103	11.27	103	11.31	098	14.54	098	14.54
	S2 C3	103	11.79	102	10.65	123	10.66	123	10.66
	S3 C0	101	10.81	099	10.13	103	13.64	103	13.64
	S3 C1	101	10.43	099	10.09	122	9.09	122	9.09
	S3 C2	101	10.81	099	10.13	103	13.64	103	13.64
	S3 C3	101	10.43	099	10.09	122	9.09	122	9.09
Multi Process Distributed Binärzähler	S0 C0	218	5.10	219	3.88	199	5.21	199	5.21
	S0 C1	220	4.73	222	4.74	216	5.37	216	5.37
	S0 C2	218	5.10	219	3.88	199	5.21	199	5.21
	S0 C3	220	4.73	222	4.74	216	5.37	216	5.37
	S1 C0	130	4.74	130	4.70	114	4.65	114	4.65
	S1 C1	130	4.55	131	5.08	125	4.99	125	4.99
	S1 C2	130	4.74	130	4.70	114	4.65	114	4.65
	S1 C3	130	4.55	131	5.08	125	4.99	125	4.99
	S2 C0	136	5.24	135	5.13	119	7.55	119	7.55
	S2 C1	136	4.59	135	4.96	141	5.91	141	5.91
	S2 C2	136	5.24	135	5.13	119	7.55	119	7.55
	S2 C3	136	4.59	135	4.96	141	5.91	141	5.91
	S3 C0	130	4.74	130	4.70	114	5.55	114	5.55
	S3 C1	130	4.55	131	5.08	124	5.45	124	5.45
	S3 C2	130	4.74	130	4.70	114	5.55	114	5.55
	S3 C3	130	4.55	131	5.08	124	5.45	124	5.45
Multi Process Distributed LFSR-Zähler	S0 C0	208	5.43	208	5.07	202	4.77	202	4.77
	S0 C1	209	5.54	210	5.03	229	4.40	229	4.40
	S0 C2	208	5.43	208	5.07	202	4.77	202	4.77
	S0 C3	209	5.54	210	5.03	229	4.40	229	4.40
	S1 C0	157	5.13	156	4.96	142	6.74	142	6.74
	S1 C1	156	4.53	155	4.38	155	5.80	155	5.80
	S1 C2	157	5.13	156	4.96	142	6.74	142	6.74
	S1 C3	156	4.53	155	4.38	155	5.80	155	5.80

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S2 C0	159	5.56	156	6.49	150	6.75	150	6.75
	S2 C1	159	6.60	157	5.87	188	4.43	188	4.43
	S2 C2	159	5.56	156	6.49	150	6.75	150	6.75
	S2 C3	159	6.60	157	5.87	188	4.43	188	4.43
	S3 C0	157	5.13	156	4.96	138	7.09	138	7.09
	S3 C1	156	4.53	155	4.38	173	5.03	173	5.03
	S3 C2	157	5.13	156	4.96	138	7.09	138	7.09
	S3 C3	156	4.53	155	4.38	173	5.03	173	5.03
Multi Process	S0 C0	218	4.28	220	4.29	221	4.15	221	4.15
Binary-Min	S0 C1	223	4.47	224	3.62	259	3.91	259	3.91
Binärzähler	S0 C2	218	4.28	220	4.29	221	4.15	221	4.15
	S0 C3	223	4.47	224	3.62	259	3.91	259	3.91
	S1 C0	125	3.02	126	3.04	107	4.38	107	4.38
	S1 C1	124	3.17	125	3.04	146	2.91	146	2.91
	S1 C2	125	3.02	126	3.04	107	4.38	107	4.38
	S1 C3	124	3.17	125	3.04	146	2.91	146	2.91
	S2 C0	156	4.64	156	4.78	140	8.67	140	8.67
	S2 C1	154	5.25	155	4.93	185	4.76	185	4.76
	S2 C2	156	4.64	156	4.78	140	8.67	140	8.67
	S2 C3	154	5.25	155	4.93	185	4.76	185	4.76
	S3 C0	125	3.02	126	3.04	110	5.28	110	5.28
	S3 C1	124	3.17	125	3.04	153	3.32	153	3.32
	S3 C2	125	3.02	126	3.04	110	5.28	110	5.28
	S3 C3	124	3.17	125	3.04	153	3.32	153	3.32
Multi Process	S0 C0	193	5.75	195	6.37	184	7.96	184	7.96
Binary-Min	S0 C1	198	5.98	197	6.61	250	4.88	250	4.88
LFSR-Zähler	S0 C2	193	5.75	195	6.37	184	7.96	184	7.96
	S0 C3	198	5.98	197	6.61	250	4.88	250	4.88
	S1 C0	148	3.19	146	3.37	137	2.51	137	2.51
	S1 C1	145	2.51	145	2.37	167	3.04	167	3.04
	S1 C2	148	3.19	146	3.37	137	2.51	137	2.51
	S1 C3	145	2.51	145	2.37	167	3.04	167	3.04
	S2 C0	156	6.15	156	5.46	147	8.28	147	8.28
	S2 C1	156	5.55	156	5.99	204	5.27	204	5.27
	S2 C2	156	6.15	156	5.46	147	8.28	147	8.28
	S2 C3	156	5.55	156	5.99	204	5.27	204	5.27
	S3 C0	148	3.19	146	3.37	140	3.70	140	3.70
	S3 C1	145	2.51	145	2.37	183	3.12	183	3.12
	S3 C2	148	3.19	146	3.37	140	3.70	140	3.70
	S3 C3	145	2.51	145	2.37	183	3.12	183	3.12
Multi Process	S0 C0	242	4.62	241	3.67	218	5.05	218	5.05
Min-Distance	S0 C1	238	4.18	237	3.62	260	3.34	260	3.34
Binärzähler	S0 C2	242	4.62	241	3.67	218	5.05	218	5.05
	S0 C3	238	4.18	237	3.62	260	3.34	260	3.34
	S1 C0	129	3.20	130	3.21	122	4.09	122	4.09
	S1 C1	128	2.92	131	2.75	141	2.92	141	2.92
	S1 C2	129	3.20	130	3.21	122	4.09	122	4.09
	S1 C3	128	2.92	131	2.75	141	2.92	141	2.92
	S2 C0	159	4.50	158	4.96	154	5.44	154	5.44
	S2 C1	159	4.53	158	4.52	231	4.82	231	4.82
	S2 C2	159	4.50	158	4.96	154	5.44	154	5.44

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S2 C3	159	4.53	158	4.52	231	4.82	231	4.82
	S3 C0	129	3.20	130	3.21	123	6.24	123	6.24
	S3 C1	128	2.92	131	2.75	154	3.12	154	3.12
	S3 C2	129	3.20	130	3.21	123	6.24	123	6.24
	S3 C3	128	2.92	131	2.75	154	3.12	154	3.12
Multi Process	S0 C0	185	4.90	187	5.48	194	6.23	194	6.23
Min-Distance	S0 C1	193	6.19	193	5.70	254	4.83	254	4.83
LFSR-Zähler	S0 C2	185	4.90	187	5.48	194	6.23	194	6.23
	S0 C3	193	6.19	193	5.70	254	4.83	254	4.83
	S1 C0	149	3.71	149	3.04	136	2.90	136	2.90
	S1 C1	148	2.71	148	2.82	174	3.05	174	3.05
	S1 C2	149	3.71	149	3.04	136	2.90	136	2.90
	S1 C3	148	2.71	148	2.82	174	3.05	174	3.05
	S2 C0	156	5.27	154	5.54	152	7.34	152	7.34
	S2 C1	158	4.63	156	5.37	210	4.63	210	4.63
	S2 C2	156	5.27	154	5.54	152	7.34	152	7.34
	S2 C3	158	4.63	156	5.37	210	4.63	210	4.63
	S3 C0	149	3.71	149	3.04	132	3.85	132	3.85
	S3 C1	148	2.71	148	2.82	177	3.38	177	3.38
	S3 C2	149	3.71	149	3.04	132	3.85	132	3.85
	S3 C3	148	2.71	148	2.82	177	3.38	177	3.38
Multi Process	S0 C0	220	3.84	221	4.16	201	5.40	201	5.40
Branch-Sequence	S0 C1	222	4.03	221	3.99	255	5.10	255	5.10
Binärzähler	S0 C2	220	3.84	221	4.16	201	5.40	201	5.40
	S0 C3	222	4.03	221	3.99	255	5.10	255	5.10
	S1 C0	121	3.14	122	3.17	103	3.87	103	3.87
	S1 C1	122	3.14	122	2.90	132	3.57	132	3.57
	S1 C2	121	3.14	122	3.17	103	3.87	103	3.87
	S1 C3	122	3.14	122	2.90	132	3.57	132	3.57
	S2 C0	146	4.13	145	4.33	126	8.51	126	8.51
	S2 C1	146	4.38	146	4.48	169	5.64	169	5.64
	S2 C2	146	4.13	145	4.33	126	8.51	126	8.51
	S2 C3	146	4.38	146	4.48	169	5.64	169	5.64
	S3 C0	121	3.14	122	3.17	096	3.86	096	3.86
	S3 C1	122	3.14	122	2.90	137	3.84	137	3.84
	S3 C2	121	3.14	122	3.17	096	3.86	096	3.86
	S3 C3	122	3.14	122	2.90	137	3.84	137	3.84
Multi Process	S0 C0	184	6.72	185	5.74	144	2.70	190	9.63
Branch-Sequence	S0 C1	189	5.54	189	5.97	245	5.36	245	5.36
LFSR-Zähler	S0 C2	184	6.72	185	5.74	190	9.63	190	9.63
	S0 C3	189	5.54	189	5.97	245	5.36	245	5.36
	S1 C0	144	3.52	144	3.99	131	3.42	131	3.42
	S1 C1	144	2.74	144	2.70	168	3.60	168	3.60
	S1 C2	144	3.52	144	3.99	131	3.42	131	3.42
	S1 C3	144	2.74	144	2.70	168	3.60	168	3.60
	S2 C0	151	5.62	150	5.68	149	7.25	149	7.25
	S2 C1	145	5.78	145	6.21	190	5.20	190	5.20
	S2 C2	151	5.62	150	5.68	149	7.25	149	7.25
	S2 C3	145	5.78	145	6.21	190	5.20	190	5.20
	S3 C0	144	3.52	144	3.99	137	4.24	137	4.24
	S3 C1	144	2.74	144	2.70	170	2.93	170	2.93

Tabelle A.4: Ergebnisse der Controllersynthese für discount

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3090A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S3 C2	144	3.52	144	3.99	137	4.24	137	4.24
	S3 C3	144	2.74	144	2.70	170	2.93	170	2.93
Multi Process	S0 C0	234	5.07	234	4.43	213	3.63	213	3.63
One-Hot	S0 C1	234	4.82	234	4.41	242	4.96	242	4.96
Binärzähler	S0 C2	234	5.07	234	4.43	213	3.63	213	3.63
	S0 C3	234	4.82	234	4.41	242	4.96	242	4.96
	S1 C0	135	3.76	135	3.92	133	6.27	133	6.27
	S1 C1	135	3.74	135	3.80	144	4.83	144	4.83
	S1 C2	135	3.76	135	3.92	133	6.27	133	6.27
	S1 C3	135	3.74	135	3.80	144	4.83	144	4.83
	S2 C0	145	4.68	145	4.82	134	7.70	144	4.83
	S2 C1	144	4.68	146	4.65	191	6.39	191	6.39
	S2 C2	145	4.68	145	4.82	133	7.04	133	7.04
	S2 C3	144	4.68	146	4.65	191	6.39	191	6.39
	S3 C0	135	3.76	135	3.92	131	6.26	131	6.26
	S3 C1	135	3.74	135	3.80	138	5.63	138	5.63
	S3 C2	135	3.76	135	3.92	131	6.26	131	6.26
	S3 C3	135	3.74	135	3.80	138	5.63	138	5.63
Multi Process	S0 C0	221	6.22	223	6.77	223	4.29	223	4.29
One Hot	S0 C1	226	5.90	228	7.64	252	4.81	252	4.81
LFSR-Zähler	S0 C2	221	6.22	223	6.77	223	4.29	223	4.29
	S0 C3	226	5.90	228	7.64	252	4.81	252	4.81
	S1 C0	160	4.08	160	3.75	152	5.25	152	5.25
	S1 C1	158	4.40	159	3.97	164	4.77	164	4.77
	S1 C2	160	4.08	160	3.75	152	5.25	152	5.25
	S1 C3	158	4.40	159	3.97	164	4.77	164	4.77
	S2 C0	167	6.41	164	5.76	154	6.50	154	6.50
	S2 C1	166	6.65	164	5.81	205	4.67	205	4.67
	S2 C2	167	6.41	164	5.76	154	6.50	154	6.50
	S2 C3	166	6.65	164	5.81	205	4.67	205	4.67
	S3 C0	160	4.08	160	3.75	153	4.57	153	4.57
	S3 C1	158	4.40	159	3.97	177	5.36	177	5.36
	S3 C2	160	4.08	160	3.75	153	4.57	153	4.57
	S3 C3	158	4.40	159	3.97	177	5.36	177	5.36

Tabelle A.4: Ergebnisse der Controllersynthese für discount

A.2 Entwurf eines Digital-Audio-Receivers

Dieser Abschnitt enthält die Ergebnisse für den in [Blinze97], [Friedr98] und [HolBli98] vorgestellten Digital-Audio-Receiver. Es wurden RTL-, Controller- und High-Level-Synthese betrachtet, wobei im Rahmen der jeweiligen Möglichkeiten die Syntheseeinstellungen variiert wurden.

A.2.1 RTL-Synthese

Es wurden die Musterlösung der Praktikumsaufgabe [Blinze97] und das RTL-Modell aus [Friedr98] benutzt. Die variierten Syntheseeinstellungen umfassen:

- Taktvorgabe
- Flächenbegrenzung (`set_max_area`)

- Strukturoptimierungen (`set_structure`, `boolean`, `timing`)
- Übersetzungsoptionen (`incremental_map` und `prioritize_min_paths`)

Die Ergebnisse der Synthese in Größe (CLBs) und FPGA-Taktrate (MHz) für den FPGA-Typ 3042-125 der beiden Modelle sind Tabelle A.6 aufgeführt, wobei die Kennzeichnung der Einstellungen gemäß Tabelle A.5 erfolgt. Die RTL-Synthese wurde mit dem Design-Compiler 1997.08 durchgeführt. Die Plazierung und die Verdrahtung des FPGAs wurde mit XACT 5.2.1 vorgenommen.

Einstellungscode	Wert	Bedeutung
T	0	keine Taktvorgabe
	1	Taktvorgabe 12MHz
	2	Taktvorgabe 16MHz = Zielfrequenz
	3	Taktvorgabe 20MHz
A	0	keine Flächenbegrenzung
	1	minimale Fläche (`set_max_area 0`)
S	0	keine Strukturierungsoption
	1	Strukturierungs-Optimierung mit Boolescher Option
	2	Strukturierungs-Optimierung mit Timing-Option
	3	Strukturierungs-Optimierung mit Boolescher und Timing-Option
C	0	`compile` ohne Zusatzoptionen
	1	`compile` mit `incremental_map`
	2	`compile` mit `prioritize_min_paths`
	3	`compile` mit `incremental_map` und `prioritize_min_paths`

Tabelle A.5: Kurzschreibweise der RTL-Syntheseeinstellungen für den DAR

Einstellung	RTL-Modell aus [Blinze97]				RTL-Modell aus [Friedr98]			
	A0		A1		A0		A1	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
T0 S0 C0	103	15.37	103	15.30	072	14.40	073	13.47
T0 S0 C1	105	16.13	105	14.90	071	14.80	073	12.77
T0 S0 C2	103	14.73	103	15.77	072	14.87	073	14.37
T0 S0 C3	105	15.20	105	15.50	071	15.40	073	13.43
T0 S1 C0	087	13.90	087	14.57	070	13.90	069	14.50
T0 S1 C1	088	16.57	089	15.87	073	15.43	072	13.20
T0 S1 C2	087	14.77	087	13.57	070	14.20	069	12.77
T0 S1 C3	088	15.33	089	17.33	073	14.43	072	12.93
T0 S2 C0	088	16.50	088	16.83	069	15.27	069	15.17
T0 S2 C1	089	18.57	089	17.67	072	14.57	073	12.63
T0 S2 C2	088	18.90	088	18.03	069	14.57	069	15.23
T0 S2 C3	089	18.30	089	18.17	072	15.33	073	13.17
T0 S3 C0	087	13.43	087	14.97	070	13.20	069	13.77
T0 S3 C1	088	14.97	089	16.60	073	15.10	072	13.47
T0 S3 C2	087	14.63	087	13.97	070	13.77	069	13.40
T0 S3 C3	088	15.97	089	16.47	073	14.57	072	12.43
T1 S0 C0	103	16.53	103	16.37	075	16.93	075	15.77
T1 S0 C1	105	17.90	105	18.43	073	14.27	073	12.67
T1 S0 C2	103	15.77	103	17.80	075	14.87	075	14.80
T1 S0 C3	105	18.13	105	17.87	073	14.23	073	13.77
T1 S1 C0	085	14.73	085	15.80	077	14.87	077	14.17

Tabelle A.6: Ergebnisse der RTL-Synthese für den Digital-Audio-Receiver

Einstellung	RTL-Modell aus [Blinze97]				RTL-Modell aus [Friedr98]			
	A0		A1		A0		A1	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
T1 S1 C1	090	15.93	090	17.03	079	13.80	079	14.03
T1 S1 C2	085	15.60	085	14.90	077	14.83	077	13.87
T1 S1 C3	090	16.93	090	16.77	079	13.97	079	15.00
T1 S2 C0	088	18.23	088	19.40	072	12.97	072	13.20
T1 S2 C1	092	19.13	092	18.60	079	15.33	079	14.90
T1 S2 C2	088	18.93	088	17.90	072	13.83	072	13.17
T1 S2 C3	092	17.87	092	19.10	079	14.33	079	14.77
T1 S3 C0	085	15.10	085	15.37	075	13.00	075	13.73
T1 S3 C1	090	15.87	090	17.50	079	14.40	079	13.60
T1 S3 C2	085	14.40	085	14.27	075	14.23	075	14.90
T1 S3 C3	090	15.80	090	16.20	079	13.90	079	14.77
T2 S0 C0	103	17.07	103	16.07	075	15.03	075	15.03
T2 S0 C1	105	18.67	105	18.57	073	14.77	073	13.67
T2 S0 C2	103	17.80	103	16.60	075	16.20	075	14.00
T2 S0 C3	105	18.40	105	19.50	073	14.07	073	14.27
T2 S1 C0	085	15.07	085	14.83	077	13.87	077	13.60
T2 S1 C1	091	18.67	091	18.27	078	14.77	078	13.93
T2 S1 C2	085	15.23	085	14.90	077	14.57	077	15.33
T2 S1 C3	091	18.93	091	19.03	078	13.83	078	14.70
T2 S2 C0	088	18.70	088	19.47	072	14.13	072	13.70
T2 S2 C1	092	17.13	092	18.90	078	14.40	078	14.67
T2 S2 C2	088	18.77	088	19.23	072	14.53	072	13.87
T2 S2 C3	092	18.50	092	18.13	078	14.57	078	14.97
T2 S3 C0	085	14.27	085	14.77	075	14.63	075	14.60
T2 S3 C1	091	19.20	091	18.60	078	14.70	078	15.00
T2 S3 C2	085	15.73	085	16.00	075	13.57	075	13.13
T2 S3 C3	091	19.20	091	18.80	078	13.93	078	13.60
T3 S0 C0	103	16.10	103	16.30	075	15.03	075	15.30
T3 S0 C1	105	18.80	105	18.53	073	13.27	073	13.13
T3 S0 C2	103	17.73	103	17.00	075	14.63	075	14.80
T3 S0 C3	105	18.53	105	18.30	073	14.03	073	13.50
T3 S1 C0	085	15.93	085	15.00	077	13.00	077	14.13
T3 S1 C1	090	18.20	090	18.20	078	14.17	078	15.10
T3 S1 C2	085	15.17	085	15.47	077	14.43	077	14.20
T3 S1 C3	090	18.77	090	19.83	078	13.23	078	15.17
T3 S2 C0	088	18.27	088	19.67	072	14.87	072	13.00
T3 S2 C1	091	19.57	091	18.97	078	15.33	078	15.33
T3 S2 C2	088	18.43	088	19.33	072	13.97	072	13.87
T3 S2 C3	091	18.37	091	19.30	078	13.37	078	15.60
T3 S3 C0	085	14.00	085	15.80	075	13.67	075	14.17
T3 S3 C1	090	18.57	090	18.23	078	14.70	078	14.27
T3 S3 C2	085	16.00	085	15.13	075	13.50	075	13.70
T3 S3 C3	090	18.40	090	18.13	078	14.73	078	13.80

Tabelle A.6: Ergebnisse der RTL-Synthese für den Digital-Audio-Receiver

A.2.2 High-Level-Synthese

Das Modell aus [Friedr98] wurde ohne Veränderung der vorgegebenen High-Level-Syntheseoptionen untersucht, da Änderungen stark negativen Einfluß auf die Syntheseergebnisse bis hin zur Undurchführbarkeit des Syntheselaufs hatten. Die variierten RTL-Syntheseeinstellungen umfassen:

- Flächenbegrenzung (`set_max_area`)
- Strukturoptimierungen (`set_structure`, `boolean`, `timing`)
- Übersetzungsoptionen (`incremental_map` und `prioritize_min_paths`)

Die Ergebnisse der Variationen zeigt Tabelle A.7, wobei die Kurzbezeichnungen aus Tabelle A.5 verwendet werden. Nach der High-Level- und RTL-Synthese mit Behavior-Compiler bzw. Design-Compiler 1997.08 erfolgte die FPGA-Abbildung mit Xilinx-XACT 5.2.1 auf ein 3190A-3-FPGA.

Optionen	High-Level-Modell aus [Friedr98] auf FPGA 3190A-3							
	C0		C1		C2		C3	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
A0 S0	162	13.40	243	10.66	162	13.64	243	10.40
A0 S1	134	15.08	233	10.58	134	15.02	233	10.46
A0 S2	124	14.70	229	11.52	124	14.60	229	11.26
A0 S3	133	14.42	233	10.48	133	14.50	233	10.30
A1 S0	162	13.42	243	10.52	162	13.20	243	10.32
A1 S1	134	14.74	233	10.86	134	14.80	233	10.32
A1 S2	124	15.04	229	11.08	124	14.42	229	11.38
A1 S3	133	15.06	233	10.44	133	14.64	233	10.32

Tabelle A.7: Ergebnisse der High-Level-Synthese für den DAR

A.2.3 Controllersynthese

Das in [HolBli98] vorgestellte Modell des DAR wurde nur mit zwei Varianten der Partitionierung und Codierung untersucht, einem globalen Distributed Code und einer manuellen Partitionierung für das Frame Send_to_DAC mit einem Binary-Min-Code. Andere Codierungen, die automatische Partitionierung sowie andere manuelle Partitionierungen waren aufgrund der Zustandskomplexität des Modells nicht für den gesamten DAR untersuchbar. Für die generierten Controller wurden die drei HDL-Modellstrukturen Single, Split und Multi Process betrachtet.

Nach der Controllersynthese mit dem Protocol-Compiler 1999.10-beta2 wurden bei der RTL-Synthese mit dem Design-Compiler 1997.08 Taktvorgabe, Flächenbegrenzung, Strukturoptimierungen und `compile`-Optionen verändert. Die Plazierung und Verdrahtung mit Xilinx-M1.5i lieferte die 384 Ergebnistupel für Schaltungsgröße (CLBs) und Geschwindigkeit (MHz) in Tabelle A.8, die sich aus den Controller-Optionen und den gemäß Tabelle A.5 kodierten RTL-Optionen ergeben.

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3064A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
Single Process Distributed	S0 C0	166	11.31	162	12.40	168	15.79	168	15.79
	S0 C1	166	13.05	162	12.85	166	12.43	166	12.43
	S0 C2	166	11.31	162	12.40	168	15.79	168	15.79
	S0 C3	166	13.05	162	12.85	166	12.43	166	12.43
	S1 C0	165	10.56	163	10.91	167	10.84	167	10.84
	S1 C1	163	11.17	161	9.27	164	15.68	164	15.68

Tabelle A.8: Ergebnisse der Controllersynthese für den DAR

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3064A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S1 C2	165	10.56	163	10.91	167	10.84	167	10.84
	S1 C3	163	11.17	161	9.27	164	15.68	164	15.68
	S2 C0	167	11.26	164	11.80	165	12.51	165	12.51
	S2 C1	167	12.41	163	10.74	164	15.38	164	15.38
	S2 C2	167	11.26	164	11.80	165	12.51	165	12.51
	S2 C3	167	12.41	163	10.74	164	15.38	164	15.38
	S3 C0	165	10.56	163	10.91	167	10.84	167	10.84
	S3 C1	163	11.17	161	9.27	164	15.68	164	15.68
	S3 C2	165	10.56	163	10.91	167	10.84	167	10.84
	S3 C3	163	11.17	161	9.27	164	15.68	164	15.68
Single Process Partitioned	S0 C0	148	12.05	146	12.21	151	16.15	151	16.15
	S0 C1	147	13.62	144	12.90	159	12.76	159	12.76
	S0 C2	148	12.05	146	12.21	151	16.15	151	16.15
	S0 C3	147	13.62	144	12.90	159	12.76	159	12.76
	S1 C0	149	11.17	148	11.62	151	11.77	151	11.77
	S1 C1	147	10.20	148	8.99	154	9.02	154	9.02
	S1 C2	149	11.17	148	11.62	151	11.77	151	11.77
	S1 C3	147	10.20	148	8.99	154	9.02	154	9.02
	S2 C0	150	12.97	148	10.45	149	14.54	149	14.54
	S2 C1	150	12.81	148	12.17	149	10.90	149	10.90
	S2 C2	150	12.97	148	10.45	149	14.54	149	14.54
	S2 C3	150	12.81	148	12.17	149	10.90	149	10.90
	S3 C0	149	11.17	148	11.62	151	11.77	151	11.77
	S3 C1	147	10.20	148	8.99	157	9.97	157	9.97
	S3 C2	149	11.17	148	11.62	151	11.77	151	11.77
	S3 C3	147	10.20	148	8.99	157	9.97	157	9.97
Split Process Distributed	S0 C0	161	11.16	161	12.96	163	16.09	163	16.09
	S0 C1	162	12.93	160	12.09	163	12.65	163	12.65
	S0 C2	161	11.16	161	12.96	163	16.09	163	16.09
	S0 C3	162	12.93	160	12.09	163	12.65	163	12.65
	S1 C0	162	13.11	160	11.02	163	15.32	163	15.32
	S1 C1	162	11.60	161	12.55	165	12.13	165	12.13
	S1 C2	162	13.11	160	11.02	163	15.32	163	15.32
	S1 C3	162	11.60	161	12.55	165	12.13	165	12.13
	S2 C0	155	11.54	155	13.00	155	14.66	155	14.66
	S2 C1	154	14.21	154	12.09	161	15.30	161	15.30
	S2 C2	155	11.54	155	13.00	155	14.66	155	14.66
	S2 C3	154	14.21	154	12.09	161	15.30	161	15.30
	S3 C0	162	13.11	160	11.02	163	15.32	163	15.32
	S3 C1	162	11.60	161	12.55	165	14.00	165	14.00
	S3 C2	162	13.11	160	11.02	163	15.32	163	15.32
	S3 C3	162	11.60	161	12.55	165	14.00	165	14.00
Split Process Partitioned	S0 C0	145	11.40	145	11.97	146	16.23	146	16.23
	S0 C1	146	13.52	146	12.77	151	14.31	151	14.31
	S0 C2	145	11.40	145	11.97	146	16.23	146	16.23
	S0 C3	146	13.52	146	12.77	151	14.31	151	14.31
	S1 C0	137	11.10	137	12.07	142	9.96	142	9.96
	S1 C1	137	12.44	137	12.34	150	13.72	150	13.72
	S1 C2	137	11.10	137	12.07	142	9.96	142	9.96
	S1 C3	137	12.44	137	12.34	150	13.72	150	13.72
	S2 C0	136	12.84	136	10.75	137	14.07	137	14.07

Tabelle A.8: Ergebnisse der Controllersynthese für den DAR

Controllertyp	Optionen	Protocol-Compiler-Modell auf FPGA 3064A-7							
		T0 A0		T0 A1		T2 A0		T2 A1	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	S2 C1	137	12.08	137	11.89	146	15.71	146	15.71
	S2 C2	136	12.84	136	10.75	137	14.07	137	14.07
	S2 C3	137	12.08	137	11.89	146	15.71	146	15.71
	S3 C0	138	10.73	138	12.79	142	9.96	142	9.96
	S3 C1	138	12.03	138	11.70	150	13.09	150	13.09
	S3 C2	138	10.73	138	12.79	142	9.96	142	9.96
	S3 C3	138	12.03	138	11.70	150	13.09	150	13.09
Multi Process Distributed	S0 C0	163	12.51	162	12.30	163	16.58	163	16.58
	S0 C1	163	12.77	162	12.34	163	15.60	163	15.60
	S0 C2	163	12.51	162	12.30	163	16.58	163	16.58
	S0 C3	163	12.77	162	12.34	163	15.60	163	15.60
	S1 C0	160	12.37	159	11.59	163	12.23	163	12.23
	S1 C1	161	11.16	160	10.89	164	14.04	164	14.04
	S1 C2	160	12.37	159	11.59	163	12.23	163	12.23
	S1 C3	161	11.16	160	10.89	164	14.04	164	14.04
	S2 C0	153	11.83	153	13.29	155	15.32	155	15.32
	S2 C1	154	11.60	153	12.04	161	16.13	161	16.13
	S2 C2	153	11.83	153	13.29	155	15.32	155	15.32
	S2 C3	154	11.60	153	12.04	161	16.13	161	16.13
	S3 C0	160	12.37	159	11.59	163	12.23	163	12.23
	S3 C1	161	11.16	160	10.89	164	13.03	164	13.03
	S3 C2	160	12.37	159	11.59	163	12.23	163	12.23
	S3 C3	161	11.16	160	10.89	164	13.03	164	13.03
Multi Process Partitioned	S0 C0	146	13.35	147	13.40	145	12.84	145	12.84
	S0 C1	147	10.96	147	11.67	151	14.45	151	14.45
	S0 C2	146	13.35	147	13.40	145	12.84	145	12.84
	S0 C3	147	10.96	147	11.67	151	14.45	151	14.45
	S1 C0	135	13.08	135	12.23	140	10.88	140	10.88
	S1 C1	135	13.00	135	13.09	153	11.23	153	11.23
	S1 C2	135	13.08	135	12.23	140	10.88	140	10.88
	S1 C3	135	13.00	135	13.09	153	11.23	153	11.23
	S2 C0	135	11.13	136	12.20	138	14.10	138	14.10
	S2 C1	136	13.58	136	10.66	146	14.35	146	14.35
	S2 C2	135	11.13	136	12.20	138	14.10	138	14.10
	S2 C3	136	13.58	136	10.66	146	14.35	146	14.35
	S3 C0	136	12.89	136	12.17	140	10.88	140	10.88
	S3 C1	136	9.79	136	11.53	153	9.68	153	9.68
	S3 C2	136	12.89	136	12.17	140	10.88	140	10.88
	S3 C3	136	9.79	136	11.53	153	9.68	153	9.68

Tabelle A.8: Ergebnisse der Controllersynthese für den DAR

A.3 Entwurf eines einfachen RISC-Prozessors

Als Grundlage für die Experimente dieses Abschnitts dienten die Musterlösung der Praktikumsaufgabe des MRISC-Prozessors in RTL-Verilog [Blinze96] und die Lösungen in RTL- und High-Level-Verilog aus [Friedr98]. Die Synthese erfolgte mit der Version 1999.05 des Design-Compiler bzw. des Behavior-Compilers, an die sich die Plazierung und Verdrahtung mit Xilinx M1.5i für ein FPGA 4010XL-3 anschloß.

A.3.1 RTL-Synthese

Die RTL-Modelle nach [Blinze96] und [Friedr98] wurden mit unterschiedlichen Einstellungen für Taktvorgabe, Flächenvorgabe, Strukturoptimierungen sowie Übersetzungsoptionen synthetisiert.

Neben den Belegungen für die RTL-Syntheseoptionen zeigt Tabelle A.9 die Kurzschreibweise, welche zur Kennzeichnung der Ergebnisse in Tabelle A.10 verwendet wird. Die Ergebnisse sind dort unterteilt nach Entwurf, Größe (CLBs) und Geschwindigkeit (MHz) dem Optionscode nachgestellt.

Einstellungscode	Wert	Bedeutung
T	0	keine Taktvorgabe
	1	Taktvorgabe 8MHz
	2	Taktvorgabe 16MHz
	3	Taktvorgabe 24MHz
	4	Taktvorgabe 33MHz
A	0	keine Flächenbegrenzung
	1	minimale Fläche (`set_max_area 0`)
S	0	keine Strukturierungsoption
	1	Strukturierungs-Optimierung mit Boolescher Option
	2	Strukturierungs-Optimierung mit Timing-Option
	3	Strukturierungs-Optimierung mit Boolescher und Timing-Option
C	0	`compile` ohne Zusatzoptionen
	1	`compile` mit `incremental_map`
	2	`compile` mit `prioritize_min_paths`
	3	`compile` mit `incremental_map` und `prioritize_min_paths`
P	0	globale Übersetzung
	1	modulweise Übersetzung

Tabelle A.9: Kurzschreibweise der RTL-Syntheseeinstellungen für den MRISC

Einstellung	RTL-Modell aus [Blinze96]				RTL-Modell aus [Friedr98]			
	A0		A1		A0		A1	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
P0 T0 S0 C0	259	12.36	259	8.91	240	10.49	240	10.30
P0 T0 S0 C1	265	8.89	260	11.92	240	10.87	239	12.91
P0 T0 S0 C2	259	12.36	259	8.91	240	10.49	240	10.30
P0 T0 S0 C3	265	8.89	260	11.92	240	10.87	239	12.91
P0 T0 S1 C0	230	8.24	230	8.24	203	14.25	203	12.43
P0 T0 S1 C1	221	8.66	222	11.83	204	12.92	202	12.85
P0 T0 S1 C2	230	8.24	230	8.24	203	14.25	203	12.43
P0 T0 S1 C3	221	8.66	222	11.83	204	12.92	202	12.85
P0 T0 S2 C0	232	11.82	232	11.82	206	12.56	205	13.31
P0 T0 S2 C1	223	8.29	223	11.72	207	12.86	205	12.99
P0 T0 S2 C2	232	11.82	232	11.82	206	12.56	205	13.31
P0 T0 S2 C3	223	8.29	223	11.72	207	12.86	205	12.99
P0 T0 S3 C0	232	9.19	232	10.15	198	12.12	196	12.45
P0 T0 S3 C1	221	8.66	222	11.83	205	12.13	204	11.62
P0 T0 S3 C2	232	9.19	232	10.15	198	12.12	196	12.45
P0 T0 S3 C3	221	8.66	222	11.83	205	12.13	204	11.62
P0 T1 S0 C0	258	15.59	259	11.83	245	12.84	246	9.70

Tabelle A.10: Ergebnisse der RTL-Synthese für den MRISC

Einstellung	RTL-Modell aus [Blinze96]				RTL-Modell aus [Friedr98]			
	A0		A1		A0		A1	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
P0 T1 S0 C1	256	10.84	257	13.48	235	10.72	230	11.72
P0 T1 S0 C2	258	15.59	259	11.83	245	12.84	246	9.70
P0 T1 S0 C3	256	10.84	257	13.48	235	10.72	230	11.72
P0 T1 S1 C0	234	10.39	234	10.90	196	11.95	196	10.97
P0 T1 S1 C1	242	13.90	242	13.42	193	14.59	195	13.56
P0 T1 S1 C2	234	10.39	234	10.90	196	11.95	196	10.97
P0 T1 S1 C3	242	13.90	242	13.42	193	14.59	195	13.56
P0 T1 S2 C0	230	10.90	229	14.68	204	14.01	203	14.44
P0 T1 S2 C1	241	12.18	242	12.71	195	14.15	194	14.51
P0 T1 S2 C2	230	10.90	229	14.68	204	14.01	203	14.44
P0 T1 S2 C3	241	12.18	242	12.71	195	14.15	194	14.51
P0 T1 S3 C0	238	11.13	238	13.48	194	13.01	194	13.01
P0 T1 S3 C1	242	13.90	242	13.42	191	13.73	194	12.74
P0 T1 S3 C2	238	11.13	238	13.48	194	13.01	194	13.01
P0 T1 S3 C3	242	13.90	242	13.42	191	13.73	194	12.74
P0 T2 S0 C0	265	16.72	267	17.36	277	16.58	280	16.44
P0 T2 S0 C1	264	17.43	265	17.19	261	18.01	269	17.51
P0 T2 S0 C2	265	16.72	267	17.36	277	16.58	280	16.44
P0 T2 S0 C3	264	17.43	265	17.19	261	18.01	269	17.51
P0 T2 S1 C0	225	17.55	225	18.82	231	17.56	229	18.20
P0 T2 S1 C1	245	17.77	248	17.75	266	18.04	266	17.66
P0 T2 S1 C2	225	17.55	225	18.82	231	17.56	229	18.20
P0 T2 S1 C3	245	17.77	248	17.75	266	18.04	266	17.66
P0 T2 S2 C0	225	17.19	225	18.52	242	17.30	241	17.72
P0 T2 S2 C1	247	17.06	247	18.45	237	17.08	235	17.14
P0 T2 S2 C2	225	17.19	225	18.52	242	17.30	241	17.72
P0 T2 S2 C3	247	17.06	247	18.45	237	17.08	235	17.14
P0 T2 S3 C0	244	18.00	243	16.20	227	17.42	227	17.13
P0 T2 S3 C1	246	16.79	246	17.84	256	16.89	254	17.15
P0 T2 S3 C2	244	18.00	243	16.20	227	17.42	227	17.13
P0 T2 S3 C3	246	16.79	246	17.84	256	16.89	254	17.15
P0 T3 S0 C0	279	19.23	279	19.12	277	16.34	276	16.84
P0 T3 S0 C1	261	18.82	263	19.15	272	20.82	283	18.93
P0 T3 S0 C2	279	19.23	279	19.12	277	16.34	276	16.84
P0 T3 S0 C3	261	18.82	263	19.15	272	20.82	283	18.93
P0 T3 S1 C0	235	25.32	235	25.11	256	20.84	265	17.92
P0 T3 S1 C1	247	19.31	249	19.11	270	21.51	272	21.27
P0 T3 S1 C2	235	25.32	235	25.11	256	20.84	265	17.92
P0 T3 S1 C3	247	19.31	249	19.11	270	21.51	272	21.27
P0 T3 S2 C0	232	24.04	233	23.78	246	22.84	249	25.11
P0 T3 S2 C1	245	21.38	246	21.57	234	22.43	235	20.65
P0 T3 S2 C2	232	24.04	233	23.78	246	22.84	249	25.11
P0 T3 S2 C3	245	21.38	246	21.57	234	22.43	235	20.65
P0 T3 S3 C0	257	23.53	256	21.49	269	20.79	267	20.42
P0 T3 S3 C1	241	18.59	242	20.21	264	21.56	264	22.32
P0 T3 S3 C2	257	23.05	256	21.49	269	20.79	267	20.42
P0 T3 S3 C3	241	18.59	242	19.30	264	21.56	264	22.32
P0 T4 S0 C0	266	17.85	266	16.23	281	13.69	281	14.82
P0 T4 S0 C1	264	17.50	263	19.21	258	18.72	265	20.10
P0 T4 S0 C2	266	17.85	266	16.23	281	13.69	281	14.82
P0 T4 S0 C3	264	17.50	263	19.21	258	18.72	265	20.10

Tabelle A.10: Ergebnisse der RTL-Synthese für den MRISC

Einstellung	RTL-Modell aus [Blinze96]				RTL-Modell aus [Friedr98]			
	A0		A1		A0		A1	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
P0 T4 S1 C0	235	22.87	235	22.93	249	16.97	249	16.55
P0 T4 S1 C1	247	16.75	244	20.93	258	18.58	260	19.71
P0 T4 S1 C2	235	22.87	235	22.93	249	16.97	249	16.55
P0 T4 S1 C3	247	16.75	245	17.87	258	18.58	260	19.71
P0 T4 S2 C0	237	23.35	238	20.50	252	21.45	252	21.11
P0 T4 S2 C1	252	23.52	251	20.57	231	21.07	235	21.39
P0 T4 S2 C2	237	23.35	238	20.50	252	21.45	252	21.11
P0 T4 S2 C3	252	23.52	251	20.57	231	21.07	235	21.39
P0 T4 S3 C0	252	20.55	250	19.12	236	18.73	237	19.50
P0 T4 S3 C1	247	16.50	247	16.45	272	21.04	272	20.40
P0 T4 S3 C2	252	20.55	250	19.12	236	18.73	237	19.50
P0 T4 S3 C3	247	16.50	247	16.45	272	21.04	272	20.40
P1 T0 S0 C0	277	9.52	276	9.29	267	9.49	265	8.87
P1 T0 S0 C1	269	10.59	261	12.12	261	10.30	260	9.40
P1 T0 S0 C2	277	9.52	276	9.29	267	9.49	264	9.86
P1 T0 S0 C3	269	10.59	261	12.12	261	10.30	260	9.40
P1 T0 S1 C0	233	12.09	233	10.36	216	11.17	217	12.52
P1 T0 S1 C1	236	11.22	236	9.57	214	12.02	211	9.95
P1 T0 S1 C2	233	12.09	233	10.36	216	11.17	217	12.52
P1 T0 S1 C3	236	11.22	236	9.57	214	12.02	211	9.95
P1 T0 S2 C0	228	12.29	229	11.15	213	10.29	213	14.04
P1 T0 S2 C1	237	13.37	237	14.07	219	10.97	222	10.00
P1 T0 S2 C2	228	12.29	229	11.15	213	10.29	213	14.04
P1 T0 S2 C3	237	13.37	237	14.07	219	10.97	222	10.00
P1 T0 S3 C0	235	11.35	235	8.97	217	12.34	217	12.65
P1 T0 S3 C1	236	11.22	236	9.57	214	12.02	211	9.95
P1 T0 S3 C2	235	11.35	235	8.97	217	12.34	217	12.65
P1 T0 S3 C3	236	11.22	236	9.57	214	12.02	211	9.95
P1 T1 S0 C0	263	11.44	262	12.45	278	9.81	285	8.66
P1 T1 S0 C1	268	11.36	263	11.38	258	12.34	254	9.46
P1 T1 S0 C2	263	11.44	262	12.45	278	9.81	285	10.44
P1 T1 S0 C3	268	11.36	263	11.38	258	12.34	254	9.46
P1 T1 S1 C0	234	10.95	238	13.63	217	13.55	217	13.87
P1 T1 S1 C1	230	11.59	230	11.28	223	11.97	213	11.42
P1 T1 S1 C2	234	10.95	238	13.63	217	13.55	217	13.87
P1 T1 S1 C3	230	11.59	230	11.28	223	11.97	213	11.42
P1 T1 S2 C0	237	14.45	230	9.69	219	10.43	215	11.68
P1 T1 S2 C1	238	10.78	238	11.05	225	13.73	220	10.23
P1 T1 S2 C2	237	14.45	230	9.69	219	10.43	215	11.68
P1 T1 S2 C3	238	10.78	238	11.05	225	13.73	220	10.23
P1 T1 S3 C0	234	13.76	238	15.22	219	12.29	219	10.59
P1 T1 S3 C1	230	11.59	230	11.28	223	11.97	213	11.42
P1 T1 S3 C2	234	13.76	238	15.22	219	12.29	219	10.59
P1 T1 S3 C3	230	11.59	230	11.28	223	11.97	213	11.42
P1 T2 S0 C0	280	17.74	281	19.47	286	13.03	289	14.51
P1 T2 S0 C1	272	18.10	271	18.99	285	16.35	271	17.89
P1 T2 S0 C2	280	17.74	281	19.47	286	13.03	289	14.67
P1 T2 S0 C3	272	18.10	271	18.99	285	16.35	271	17.89
P1 T2 S1 C0	234	16.94	237	19.04	234	19.16	232	18.11
P1 T2 S1 C1	236	18.64	231	19.07	247	16.86	236	18.49
P1 T2 S1 C2	234	16.94	237	19.04	234	19.16	232	18.11

Tabelle A.10: Ergebnisse der RTL-Synthese für den MRISC

Einstellung	RTL-Modell aus [Blinze96]				RTL-Modell aus [Friedr98]			
	A0		A1		A0		A1	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
P1 T2 S1 C3	236	18.64	231	19.07	247	16.86	236	18.49
P1 T2 S2 C0	246	16.68	230	17.87	260	17.04	240	17.82
P1 T2 S2 C1	237	18.06	234	18.27	250	16.73	244	17.17
P1 T2 S2 C2	246	16.68	230	17.87	260	17.04	240	17.82
P1 T2 S2 C3	237	18.06	234	18.27	250	16.73	244	17.17
P1 T2 S3 C0	243	17.81	244	17.28	230	18.13	228	17.07
P1 T2 S3 C1	236	18.64	231	19.07	247	16.86	236	18.49
P1 T2 S3 C2	243	17.81	244	17.28	230	18.43	228	19.20
P1 T2 S3 C3	236	18.64	231	19.07	247	16.86	236	18.49
P1 T3 S0 C0	270	17.77	270	19.77	279	12.56	289	12.22
P1 T3 S0 C1	270	20.02	261	21.12	286	19.38	270	17.38
P1 T3 S0 C2	270	17.77	270	19.77	279	12.56	289	12.74
P1 T3 S0 C3	270	20.02	261	21.12	286	19.38	270	17.38
P1 T3 S1 C0	234	22.94	237	21.28	236	21.45	237	22.30
P1 T3 S1 C1	234	18.98	241	20.57	258	22.34	266	16.65
P1 T3 S1 C2	234	22.94	237	21.28	236	21.45	237	22.30
P1 T3 S1 C3	234	18.98	241	20.57	258	22.34	266	16.65
P1 T3 S2 C0	254	21.16	245	22.16	238	23.11	225	20.86
P1 T3 S2 C1	243	22.41	234	19.07	261	20.18	260	17.82
P1 T3 S2 C2	254	21.16	245	22.16	238	22.58	225	20.86
P1 T3 S2 C3	243	22.41	234	19.07	261	20.18	260	17.82
P1 T3 S3 C0	240	17.68	244	19.82	229	21.61	229	21.48
P1 T3 S3 C1	234	18.98	241	20.57	258	22.34	266	16.65
P1 T3 S3 C2	240	17.68	244	19.82	229	21.61	229	21.29
P1 T3 S3 C3	234	18.98	241	20.57	258	22.34	266	16.65
P1 T4 S0 C0	270	18.85	270	23.27	286	9.52	285	9.15
P1 T4 S0 C1	269	25.11	262	22.91	285	17.25	278	16.68
P1 T4 S0 C2	270	18.85	270	23.27	286	9.52	285	9.15
P1 T4 S0 C3	269	25.11	262	22.91	285	17.25	278	16.68
P1 T4 S1 C0	237	17.60	239	23.09	235	21.61	233	19.80
P1 T4 S1 C1	235	18.98	235	25.15	251	20.34	234	17.42
P1 T4 S1 C2	237	17.60	239	23.09	235	21.61	233	19.80
P1 T4 S1 C3	235	18.98	235	25.15	251	20.34	234	17.42
P1 T4 S2 C0	248	19.06	233	18.84	244	21.38	266	20.20
P1 T4 S2 C1	245	21.28	238	21.80	258	15.92	260	16.64
P1 T4 S2 C2	248	19.06	233	18.84	244	21.38	266	20.20
P1 T4 S2 C3	245	18.57	238	21.80	258	15.92	260	16.64
P1 T4 S3 C0	231	20.68	237	22.03	229	20.58	229	21.47
P1 T4 S3 C1	235	18.98	235	25.15	251	20.34	234	17.42
P1 T4 S3 C2	231	20.68	237	22.03	229	18.53	229	20.60
P1 T4 S3 C3	235	18.98	235	25.15	251	20.34	234	17.42

Tabelle A.10: Ergebnisse der RTL-Synthese für den MRISC

Zur genaueren Untersuchung des Einflusses der Taktvorgabe auf die Größe und die Geschwindigkeit der RTL-Syntheseergebnisse wurde die Taktvorgabe in einem weiteren Experiment von 0,2 bis 40MHz in Schritten zu 200kHz variiert. Neben einer minimalen Flächenvorgabe wurden dabei beide Strukturoptimierungen und keine `compile`-Optionen benutzt. Die Ergebnisse hierzu zeigt Tabelle A.11.

Vorgabe MHz	[Blinze96]		[Friedr98]		Vorgabe MHz	[Blinze96]		[Friedr98]	
	CLBs	MHz	CLBs	MHz		CLBs	MHz	CLBs	MHz
0.2	238	12.28	194	12.27	20.6	243	21.37	244	20.62
0.4	238	11.58	194	12.24	20.8	242	21.59	244	20.16
0.6	238	12.11	194	12.61	21.0	243	21.38	244	20.89
0.8	238	11.58	194	13.13	21.2	245	21.37	234	19.69
1.0	238	12.26	194	10.47	21.4	243	22.05	234	20.55
1.2	238	11.49	194	11.57	21.6	243	22.38	234	21.19
1.4	238	11.32	194	10.85	21.8	254	20.75	231	20.99
1.6	238	11.36	194	12.07	22.0	254	22.29	234	21.78
1.8	238	9.76	194	11.69	22.2	254	22.20	234	21.40
2.0	238	12.98	194	11.53	22.4	256	23.23	234	21.24
2.2	238	9.33	194	11.74	22.6	255	22.82	234	21.41
2.4	238	10.41	194	12.33	22.8	256	22.57	234	20.44
2.6	238	10.73	194	11.60	23.0	238	23.18	255	21.46
2.8	238	9.21	194	11.38	23.2	238	23.47	256	20.47
3.0	238	12.07	194	11.54	23.4	238	23.85	267	19.75
3.2	238	10.71	194	11.38	23.6	238	23.76	267	21.23
3.4	238	12.71	194	10.44	23.8	241	24.21	267	19.96
3.6	238	10.31	194	13.10	24.0	241	22.92	267	21.34
3.8	238	12.61	194	11.46	24.2	241	23.25	267	21.67
4.0	238	10.85	194	11.48	24.4	241	24.52	267	19.77
4.2	238	11.87	194	10.79	24.6	246	22.13	267	19.51
4.4	238	12.91	194	11.39	24.8	256	23.65	267	21.54
4.6	238	9.22	194	10.87	25.0	256	21.49	267	20.42
4.8	238	10.72	194	11.17	25.2	241	24.35	267	21.65
5.0	238	11.49	194	13.61	25.4	260	21.05	234	21.09
5.2	238	10.50	194	11.12	25.6	241	25.61	234	20.73
5.4	238	12.48	194	11.78	25.8	241	24.20	233	19.97
5.6	238	10.73	194	10.93	26.0	241	23.75	252	19.62
5.8	238	11.79	194	12.89	26.2	257	24.38	252	20.69
6.0	238	10.70	194	11.96	26.4	245	23.45	252	21.31
6.2	238	12.46	194	11.47	26.6	246	23.98	244	20.75
6.4	238	11.59	194	11.80	26.8	246	24.31	244	22.38
6.6	238	10.34	194	10.78	27.0	247	24.89	252	19.78
6.8	238	14.20	194	12.05	27.2	246	21.60	244	22.07
7.0	238	10.29	194	11.38	27.4	244	20.69	252	19.16
7.2	238	12.66	194	12.24	27.6	249	20.81	244	23.03
7.4	238	11.50	193	11.88	27.8	249	21.09	252	20.62
7.6	238	11.82	194	13.18	28.0	243	21.67	252	21.18
7.8	238	11.78	194	10.09	28.2	242	23.34	235	19.97
8.0	238	13.48	194	13.01	28.4	240	23.32	235	19.21
8.2	238	13.45	194	11.67	28.6	247	22.37	235	18.36
8.4	238	13.52	201	11.66	28.8	240	17.80	235	19.75
8.6	238	11.36	203	11.98	29.0	240	19.55	259	20.15
8.8	238	16.93	203	10.92	29.2	239	17.07	235	18.83
9.0	238	13.12	202	11.22	29.4	241	17.43	256	20.60
9.2	238	14.20	203	12.29	29.6	241	18.34	235	20.69
9.4	238	14.96	202	12.55	29.8	241	19.62	253	18.50
9.6	238	15.88	203	13.06	30.0	238	19.46	236	17.62
9.8	238	16.74	200	12.03	30.2	238	18.53	236	19.13
10.0	238	12.97	203	13.44	30.4	251	16.94	236	18.56

Tabelle A.11: Ergebnisse der taktgesteuerten RTL-Synthese für den MRISC

Vorgabe MHz	[Blinze96]		[Friedr98]		Vorgabe MHz	[Blinze96]		[Friedr98]	
	CLBs	MHz	CLBs	MHz		CLBs	MHz	CLBs	MHz
10.2	238	15.52	204	12.71	30.6	251	18.19	236	18.68
10.4	239	14.78	200	13.18	30.8	241	21.31	236	20.58
10.6	238	11.81	205	14.14	31.0	241	19.15	236	19.36
10.8	238	16.68	203	15.02	31.2	251	18.79	236	18.16
11.0	227	14.33	206	14.30	31.4	254	18.70	236	18.12
11.2	227	14.13	208	14.01	31.6	244	24.35	237	19.68
11.4	227	13.09	207	14.95	31.8	244	23.56	237	18.32
11.6	227	15.73	205	14.96	32.0	244	23.05	237	19.55
11.8	227	15.50	212	13.99	32.2	252	23.18	237	17.57
12.0	227	17.12	224	14.41	32.4	239	22.17	237	19.58
12.2	227	16.00	218	12.39	32.6	254	20.08	237	17.99
12.4	241	17.83	215	14.56	32.8	253	20.49	237	16.15
12.6	241	16.56	227	15.55	33.0	254	20.77	237	16.81
12.8	241	18.49	226	15.46	33.2	246	18.28	237	18.06
13.0	242	15.24	226	15.50	33.4	250	20.80	237	19.30
13.2	242	15.34	231	16.42	33.6	254	19.51	237	19.91
13.4	242	17.27	236	14.59	33.8	249	21.57	237	18.79
13.6	242	16.05	240	16.30	34.0	246	21.81	237	18.23
13.8	242	17.86	225	15.42	34.2	251	22.11	237	17.88
14.0	242	15.63	227	15.76	34.4	250	18.00	237	16.71
14.2	243	16.77	229	17.20	34.6	240	21.87	237	18.38
14.4	242	16.51	236	17.49	34.8	246	18.60	237	16.32
14.6	242	17.63	244	15.90	35.0	246	22.36	237	17.03
14.8	242	17.86	240	15.65	35.2	246	17.96	237	17.43
15.0	242	16.84	261	16.40	35.4	245	23.42	237	18.10
15.2	242	16.36	249	16.26	35.6	244	17.46	237	16.70
15.4	242	17.68	251	16.60	35.8	250	24.39	237	19.13
15.6	242	16.68	226	16.76	36.0	247	21.13	237	18.63
15.8	242	17.22	226	16.68	36.2	245	16.16	237	17.69
16.0	242	17.61	227	17.64	36.4	240	24.95	237	18.35
16.2	243	17.73	227	17.51	36.6	240	20.58	237	18.85
16.4	242	17.82	256	17.88	36.8	240	18.60	237	17.86
16.6	244	17.20	251	17.49	37.0	240	19.31	237	17.01
16.8	243	18.36	244	18.35	37.2	240	19.91	237	16.66
17.0	243	19.32	265	18.41	37.4	240	23.85	237	17.30
17.2	244	19.74	273	17.85	37.6	240	19.07	237	19.41
17.4	242	18.98	230	18.57	37.8	240	21.16	237	16.52
17.6	242	20.32	230	18.28	38.0	239	23.47	237	17.70
17.8	242	18.31	230	17.80	38.2	239	24.27	237	17.68
18.0	242	18.70	230	19.28	38.4	240	23.34	237	17.20
18.2	243	20.01	231	18.26	38.6	240	22.02	237	17.37
18.4	244	20.31	230	18.61	38.8	240	23.65	237	17.49
18.6	243	19.62	226	19.03	39.0	245	19.21	225	15.63
18.8	244	20.63	241	19.16	39.2	244	21.99	240	16.40
19.0	244	19.73	238	19.32	39.4	250	18.73	237	15.77
19.2	244	20.85	240	19.38	39.6	241	19.58	237	16.16
19.4	244	20.56	240	20.32	39.8	244	15.82	237	16.99
19.6	244	19.78	240	20.19	40.0	251	19.14	237	15.48
19.8	240	20.58	240	19.93	40.2	248	18.43	237	15.62
20.0	240	20.12	244	20.14	40.4	252	20.71	237	15.76
20.2	244	21.46	244	20.51	40.6	243	23.51	237	17.23
20.4	245	20.83	244	20.44	40.8	245	15.99	237	18.40

Tabelle A.11: Ergebnisse der taktgesteuerten RTL-Synthese für den MRISC

A.3.2 High-Level-Synthese

Das High-Level-Modell des MRISC aus [Friedr98] wurde mit unterschiedlichen Taktvorgaben für die High-Level-Synthese sowie verschiedenen Kombinationen der RTL-Syntheseoptionen für die Flächenvorgabe, die Strukturoptimierungen und die Übersetzungsoptionen untersucht.

Tabelle A.12 gibt einen Überblick über die Belegungen der High-Level- und RTL-Syntheseoptionen sowie der den Optionen zugeordneten Kurzbeschreibung für die in Tabelle A.13 zusammengefaßten Syntheseergebnisse.

Einstellungscode	Wert	Bedeutung
T	1	Taktvorgabe 2.5MHz
	2	Taktvorgabe 5MHz
	3	Taktvorgabe 10MHz
	4	Taktvorgabe 20MHz
	5	Taktvorgabe 30MHz
A	0	keine Flächenbegrenzung
	1	minimale Fläche (`set_max_area 0`)
S	0	keine Strukturierungsoption
	1	Strukturierungs-Optimierung mit Boolescher Option
	2	Strukturierungs-Optimierung mit Timing-Option
	3	Strukturierungs-Optimierung mit Boolescher und Timing-Option
C	0	`compile` ohne Zusatzoptionen
	1	`compile` mit `incremental_map`
	2	`compile` mit `prioritize_min_paths`
	3	`compile` mit `incremental_map` und `prioritize_min_paths`

Tabelle A.12: HL/RTL-Syntheseeinstellungen für den MRISC

Optionen	C0		C1		C2		C3	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
T1 A0 S0	222	12.14	263	11.08	222	12.14	263	11.08
T1 A0 S1	222	12.14	263	11.08	222	12.14	263	11.08
T1 A0 S2	222	12.14	263	11.08	222	12.14	263	11.08
T1 A0 S3	222	12.14	263	11.08	222	12.14	263	11.08
T1 A1 S0	222	9.74	261	12.04	222	9.74	261	12.04
T1 A1 S1	222	9.74	261	12.04	222	9.74	261	12.04
T1 A1 S2	222	9.74	261	12.04	222	9.74	261	12.04
T1 A1 S3	222	9.74	261	12.04	222	9.74	261	12.04
T2 A0 S0	222	11.40	263	11.67	222	11.40	263	11.67
T2 A0 S1	222	11.40	263	11.67	222	11.40	263	11.67
T2 A0 S2	222	11.40	263	11.67	222	11.40	263	11.67
T2 A0 S3	222	11.40	263	11.67	222	11.40	263	11.67
T2 A1 S0	222	10.77	261	10.46	222	10.77	261	10.46
T2 A1 S1	222	10.77	261	10.46	222	10.77	261	10.46
T2 A1 S2	222	10.77	261	10.46	222	10.77	261	10.46
T2 A1 S3	222	10.77	261	10.46	222	10.77	261	10.46
T3 A0 S0	221	12.52	274	13.69	221	12.52	274	13.69
T3 A0 S1	221	12.52	274	13.69	221	12.52	274	13.69
T3 A0 S2	221	12.52	274	13.69	221	12.52	274	13.69
T3 A0 S3	221	12.52	274	13.69	221	12.52	274	13.69

Tabelle A.13: Ergebnisse der High-Level-Synthese für MRISC

Optionen	C0		C1		C2		C3	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
T3 A1 S0	221	13.65	272	13.01	221	13.65	272	13.01
T3 A1 S1	221	13.65	272	13.01	221	13.65	272	13.01
T3 A1 S2	221	13.65	272	13.01	221	13.65	272	13.01
T3 A1 S3	221	13.65	272	13.01	221	13.65	272	13.01
T4 A0 S0	258	21.35	340	20.73	258	21.35	340	20.73
T4 A0 S1	258	21.35	340	20.73	258	21.35	340	20.73
T4 A0 S2	258	21.35	340	20.73	258	21.35	340	20.73
T4 A0 S3	258	21.35	340	20.73	258	21.35	340	20.73
T4 A1 S0	259	20.67	339	20.11	259	20.67	339	20.11
T4 A1 S1	259	20.67	339	20.11	259	20.67	339	20.11
T4 A1 S2	259	20.67	339	20.11	259	20.67	339	20.11
T4 A1 S3	259	20.67	339	20.11	259	20.67	339	20.11
T5 A0 S0	335	27.42	388	23.10	335	27.42	388	23.10
T5 A0 S1	335	27.42	388	23.10	335	27.42	388	23.10
T5 A0 S2	335	27.42	388	23.10	335	27.42	388	23.10
T5 A0 S3	335	27.42	388	23.10	335	27.42	388	23.10
T5 A1 S0	334	27.09	385	26.23	334	27.09	385	26.23
T5 A1 S1	334	27.09	385	26.23	334	27.09	385	26.23
T5 A1 S2	334	27.09	385	26.23	334	27.09	385	26.23
T5 A1 S3	334	27.09	385	26.23	334	27.09	385	26.23

Tabelle A.13: Ergebnisse der High-Level-Synthese für MRISC

Der Einfluß der Taktvorgabe wurde in einem weiteren Experiment durch Variation des Taktes von 0,2 bis 40MHz in Schritten zu 200KHz genauer untersucht, wobei eine minimale Flächen und beide Strukturoptimierungen vorgegeben wurden. Die `compile`-Option `incremental_map` wurde an- und abgeschaltet, während `min_paths` deaktiviert war. Die Ergebnisse diese Experimentes sind in Tabelle A.14 aufgelistet.

Vorgabe MHz	incremental off		incremental on		Vorgabe MHz	incremental off		incremental on	
	CLBs	MHz	CLBs	MHz		CLBs	MHz	CLBs	MHz
0.2	222	12.37	260	10.47	20.6	260	20.66	344	20.67
0.4	222	12.92	260	10.29	20.8	260	21.30	331	20.97
0.6	222	12.22	260	10.16	21.0	260	21.78	321	18.85
0.8	222	12.14	260	10.45	21.2	274	19.77	395	21.25
1.0	222	11.86	260	10.94	21.4	276	22.36	370	21.48
1.2	222	11.32	260	9.81	21.6	274	22.31	359	19.84
1.4	222	12.56	260	10.67	21.8	274	22.13	359	21.07
1.6	222	12.43	260	11.15	22.0	281	22.62	367	18.20
1.8	222	10.24	260	11.05	22.2	285	22.25	362	17.16
2.0	222	12.03	260	10.00	22.4	282	22.52	362	17.37
2.2	222	12.21	260	10.59	22.6	282	23.10	362	17.69
2.4	222	9.96	260	11.56	22.8	282	22.88	362	17.77
2.6	222	9.90	260	11.24	23.0	268	21.67	348	15.85
2.8	222	10.02	260	11.04	23.2	271	23.55	348	17.81
3.0	222	10.75	260	11.46	23.4	267	21.80	334	17.77
3.2	222	8.86	260	10.95	23.6	277	22.95	332	18.51
3.4	222	11.23	260	10.72	23.8	277	22.41	332	18.03
3.6	222	10.71	260	10.52	24.0	277	23.24	331	19.17
3.8	222	10.67	260	11.40	24.2	296	24.51	330	23.95

Tabelle A.14: Ergebnisse der taktgesteuerten High-Level-Synthese für MRISC

Vorgabe MHz	incremental off		incremental on		Vorgabe MHz	incremental off		incremental on	
	CLBs	MHz	CLBs	MHz		CLBs	MHz	CLBs	MHz
4.0	222	12.52	260	10.16	24.4	296	24.50	361	21.14
4.2	222	9.79	260	10.49	24.6	296	24.93	331	23.59
4.4	222	10.45	260	10.40	24.8	287	23.41	350	25.48
4.6	222	11.44	260	10.47	25.0	287	24.16	354	24.22
4.8	222	10.99	260	9.17	25.2	-	-	-	-
5.0	222	10.77	260	11.96	25.4	-	-	-	-
5.2	222	11.60	260	11.24	25.6	-	-	-	-
5.4	222	12.46	260	11.16	25.8	-	-	-	-
5.6	222	11.45	260	10.99	26.0	-	-	-	-
5.8	222	10.58	260	11.01	26.2	-	-	-	-
6.0	222	10.39	261	11.66	26.4	-	-	-	-
6.2	222	11.44	265	11.49	26.6	-	-	-	-
6.4	221	11.50	253	12.61	26.8	-	-	-	-
6.6	224	12.52	266	12.33	27.0	-	-	-	-
6.8	223	13.60	266	12.36	27.2	-	-	-	-
7.0	224	8.78	249	11.07	27.4	-	-	-	-
7.2	223	12.56	251	12.21	27.6	-	-	-	-
7.4	224	11.76	251	10.70	27.8	-	-	-	-
7.6	224	9.94	254	11.31	28.0	-	-	-	-
7.8	308	12.22	362	11.95	28.2	-	-	-	-
8.0	222	11.35	271	13.19	28.4	-	-	-	-
8.2	222	14.44	264	11.32	28.6	-	-	-	-
8.4	223	12.70	272	11.41	28.8	-	-	-	-
8.6	224	12.69	262	10.99	29.0	-	-	-	-
8.8	223	11.59	264	12.16	29.2	-	-	-	-
9.0	221	12.81	276	12.14	29.4	-	-	-	-
9.2	225	10.47	264	11.76	29.6	-	-	-	-
9.4	223	14.71	278	11.71	29.8	-	-	-	-
9.6	223	12.36	265	12.84	30.0	323	30.00	362	23.36
9.8	221	13.00	269	13.27	30.2	323	30.33	362	24.20
10.0	221	13.65	273	12.52	30.4	328	27.36	352	29.04
10.2	220	13.65	277	13.01	30.6	322	29.67	352	28.29
10.4	222	13.51	274	12.71	30.8	322	30.83	352	27.48
10.6	221	14.02	271	12.92	31.0	322	29.01	353	23.77
10.8	219	14.03	290	12.72	31.2	326	28.69	353	28.89
11.0	223	15.10	285	12.34	31.4	309	24.45	326	28.54
11.2	224	14.44	298	12.87	31.6	302	25.86	326	27.40
11.4	225	13.57	291	12.41	31.8	302	25.21	326	28.59
11.6	225	15.28	293	14.82	32.0	302	28.13	326	30.05
11.8	225	14.10	299	13.27	32.2	302	27.19	326	31.43
12.0	226	15.19	297	15.22	32.4	302	29.27	326	31.19
12.2	228	14.06	304	14.55	32.6	302	26.66	326	26.01
12.4	219	14.86	295	13.99	32.8	302	25.85	326	28.50
12.6	224	14.35	322	14.16	33.0	302	25.88	326	28.23
12.8	224	15.67	314	14.59	33.2	302	23.45	326	31.95
13.0	229	14.65	313	14.97	33.4	302	27.77	326	29.70
13.2	225	17.13	313	15.71	33.6	302	25.70	326	29.19
13.4	233	16.17	320	16.80	33.8	302	26.22	326	31.47
13.6	229	15.43	304	15.10	34.0	302	28.52	326	29.23
13.8	253	16.86	337	15.69	34.2	279	27.20	338	26.47
14.0	242	15.67	342	15.75	34.4	279	27.58	337	28.23
14.2	237	18.57	348	16.24	34.6	279	27.90	338	29.57

Tabelle A.14: Ergebnisse der taktgesteuerten High-Level-Synthese für MRISC

Vorgabe MHz	incremental off		incremental on		Vorgabe MHz	incremental off		incremental on	
	CLBs	MHz	CLBs	MHz		CLBs	MHz	CLBs	MHz
14.4	244	17.38	354	16.10	34.8	279	27.40	337	24.15
14.6	290	16.86	400	15.70	35.0	279	29.03	336	28.97
14.8	242	17.13	332	16.07	35.2	279	29.25	336	27.54
15.0	244	17.99	325	16.86	35.4	279	28.45	336	23.67
15.2	244	17.98	325	15.74	35.6	286	32.39	334	29.02
15.4	241	18.21	325	16.53	35.8	286	30.43	334	29.43
15.6	246	18.75	331	16.49	36.0	286	28.25	338	29.49
15.8	247	18.43	326	17.57	36.2	286	30.77	338	30.43
16.0	243	18.39	316	17.01	36.4	286	29.05	338	30.34
16.2	242	18.05	316	17.46	36.6	294	29.28	338	29.87
16.4	242	17.83	328	17.25	36.8	286	27.34	338	29.98
16.6	259	20.47	328	17.59	37.0	286	28.94	334	27.98
16.8	259	19.34	328	18.25	37.2	288	30.55	333	26.93
17.0	248	19.27	322	17.30	37.4	288	28.41	333	31.92
17.2	263	19.52	338	17.66	37.6	288	30.03	333	27.68
17.4	241	22.52	348	18.58	37.8	297	29.44	333	30.02
17.6	241	18.65	318	19.43	38.0	297	28.01	333	29.72
17.8	241	21.12	318	18.77	38.2	297	27.87	333	28.77
18.0	262	19.26	356	18.95	38.4	297	25.69	333	29.75
18.2	262	20.16	355	19.60	38.6	297	30.09	333	29.33
18.4	262	20.01	355	19.23	38.8	297	26.49	333	29.12
18.6	262	20.29	353	19.27	39.0	296	26.99	333	28.11
18.8	253	20.63	356	19.57	39.2	289	27.23	338	29.89
19.0	261	20.60	349	20.56	39.4	289	28.32	338	28.62
19.2	257	20.99	330	20.99	39.6	296	28.89	338	29.83
19.4	257	21.32	330	20.39	39.8	289	30.72	338	29.30
19.6	257	20.45	326	19.87	40.0	296	30.21	338	28.86
19.8	259	21.28	320	20.33	40.2	289	29.59	339	30.07
20.0	259	20.67	338	20.04	40.4	296	29.50	333	28.71
20.2	259	20.77	338	21.08	40.6	289	30.56	335	27.81
20.4	260	22.47	338	20.88	40.8	289	30.78	334	30.23

Tabelle A.14: Ergebnisse der taktgesteuerten High-Level-Synthese für MRISC

A.4 Entwurf eines Chipkartenlesers

Im folgenden werden die Syntheseergebnisse des Chipkartenlesers [Blinze99A] für die RTL- und die Controllersynthese in den möglichen Variationen der jeweiligen Syntheseparameter betrachtet. Für die RTL-Syntheseschritte konnten dabei dieselben Optionsvarianten angewendet werden, deren Kurzbeschreibung in den Tabellen der RTL- bzw. Controllersynthese entsprechend Tabelle A.15 aufgebaut ist. Als Zielhardware wurde in beiden Synthesearten ein Xilinx 4013E-4 FPGA benutzt.

Einstellungscode	Wert	Bedeutung
T	0	keine Taktvorgabe
	1	Taktvorgabe 0,5MHz
	2	Taktvorgabe 1MHz
	3	Taktvorgabe 2MHz
A	0	keine Flächenbegrenzung
	1	minimale Fläche (`set_max_area 0`)

Tabelle A.15: RTL-Syntheseeinstellungen für den Chipkartenleser

Einstellungscode	Wert	Bedeutung
S	0	keine Strukturierungsoption
	1	Strukturierungs-Optimierung mit Boolescher Option
	2	Strukturierungs-Optimierung mit Timing-Option
	3	Strukturierungs-Optimierung mit Boolescher und Timing-Option
C	0	`compile` ohne Zusatzoptionen
	1	`compile` mit `incremental_map`
	2	`compile` mit `prioritize_min_paths`
	3	`compile` mit `incremental_map` und `prioritize_min_paths`

Tabelle A.15: RTL-Syntheseeinstellungen für den Chipkartenleser

A.4.1 RTL-Synthese

Die Synthese des RTL-Modells erfolgte mit dem Design-Compiler 1999.05 und die anschließende Plazierung und Verdrahtung des FPGAs mit Xilinx M1.5i. Die dabei entstandenen Ergebnisse zeigt Tabelle A.16.

Optionen	C0		C1		C2		C3	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
T0 A0 S0	455	9.37	458	8.89	455	9.37	458	8.89
T0 A0 S1	374	8.28	376	9.00	374	8.28	376	9.00
T0 A0 S2	398	8.51	398	7.93	398	8.51	398	7.93
T0 A0 S3	374	8.20	379	9.21	374	8.20	379	9.21
T0 A1 S0	450	9.77	451	11.09	450	9.77	451	11.09
T0 A1 S1	373	8.70	373	8.95	373	8.70	373	8.95
T0 A1 S2	400	8.50	396	8.63	400	8.50	396	8.63
T0 A1 S3	373	9.16	378	9.14	373	9.16	378	9.14
T1 A0 S0	432	8.79	434	9.41	432	8.79	434	9.41
T1 A0 S1	345	9.24	353	8.77	345	9.24	353	8.77
T1 A0 S2	367	8.85	377	8.75	367	8.85	377	8.75
T1 A0 S3	344	8.94	353	9.74	344	8.94	353	9.74
T1 A1 S0	428	10.78	429	10.91	428	10.78	429	10.91
T1 A1 S1	345	8.65	353	9.35	345	8.65	353	9.35
T1 A1 S2	366	9.87	379	8.86	366	9.87	379	8.86
T1 A1 S3	344	9.26	354	9.88	344	9.22	354	9.88
T2 A0 S0	432	9.82	434	9.09	432	9.82	434	9.09
T2 A0 S1	345	9.41	353	8.46	345	9.41	353	8.46
T2 A0 S2	367	8.97	377	8.50	367	8.97	377	8.50
T2 A0 S3	344	8.93	353	9.51	344	8.93	353	9.51
T2 A1 S0	428	11.77	429	9.73	428	11.77	429	9.73
T2 A1 S1	345	8.48	353	9.59	345	8.48	353	9.59
T2 A1 S2	366	9.84	379	9.07	366	9.84	379	9.07
T2 A1 S3	344	8.99	354	9.87	344	8.83	354	9.87
T3 A0 S0	432	8.73	434	8.93	432	8.73	434	8.93
T3 A0 S1	345	9.21	353	8.94	345	9.21	353	8.94
T3 A0 S2	367	9.08	377	8.68	367	9.08	377	8.68
T3 A0 S3	344	8.75	353	9.52	344	8.75	353	9.52
T3 A1 S0	428	11.98	429	10.51	428	11.98	429	10.51
T3 A1 S1	345	8.74	353	9.57	345	8.74	353	9.57
T3 A1 S2	366	9.79	379	8.92	366	9.79	379	8.92
T3 A1 S3	344	9.06	354	9.78	344	8.98	354	9.78

Tabelle A.16: Ergebnisse der RTL-Synthese des Chipkartenlesers

Der Einfluß der Taktvorgabe wurde in einer weiteren Versuchsreihe mit Vorgaben von 0,5 bis 16,5MHz in Schritten zu 500kHz untersucht. Hierbei wurden neben der minimalen Flächenvorgabe beide Strukturoptimierungen und keine `compile`-Optionen aktiviert (Tabelle A.17).

Vorgabe MHz	Ergebnis		Vorgabe MHz	Ergebnis		Vorgabe MHz	Ergebnis	
	CLBs	MHz		CLBs	MHz		CLBs	MHz
0.5	344	9.22	6.0	346	8.46	11.5	352	11.74
1.0	344	8.83	6.5	348	8.70	12.0	354	12.10
1.5	344	8.99	7.0	349	10.07	12.5	354	12.46
2.0	344	8.98	7.5	349	9.72	13.0	353	12.20
2.5	344	7.81	8.0	352	9.84	13.5	353	12.66
3.0	344	8.95	8.5	352	9.68	14.0	353	12.21
3.5	344	8.61	9.0	351	10.77	14.5	354	11.54
4.0	344	8.49	9.5	351	10.71	15.0	353	12.15
4.5	344	9.20	10.0	350	11.07	15.5	354	12.00
5.0	344	9.20	10.5	352	11.88	16.0	354	11.54
5.5	344	9.54	11.0	352	12.47	16.5	353	9.73

Tabelle A.17: Ergebnisse der taktgesteuerten RTL-Synthese des Chipkartenlesers

A.4.2 Controllersynthese

Die Controllersynthese erfolgte mit dem Protocol-Compiler 1999.10-beta2 und dem Design-Compiler 1999.05. Die Plazierung und Verdrahtung des FPGAs wurde mit Xilinx M1.5i durchgeführt. Tabelle A.18 faßt die dabei entstandenden Ergebnisse zusammen.

Controllertyp	Optionen	T0		T1		T2		T3	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
Single Process Automatic	A0 S0 C0	352	8.36	349	8.60	349	8.99	349	9.89
	A0 S0 C1	355	9.21	351	10.19	351	9.83	351	10.39
	A0 S0 C2	352	8.36	349	8.60	349	8.99	349	9.89
	A0 S0 C3	355	9.21	351	10.19	351	9.83	351	10.39
	A0 S1 C0	317	8.39	286	8.51	286	8.32	286	8.80
	A0 S1 C1	315	8.35	299	9.19	299	9.34	299	9.15
	A0 S1 C2	317	8.39	286	8.51	286	8.32	286	8.80
	A0 S1 C3	315	8.35	299	9.19	299	9.34	299	9.15
	A0 S2 C0	328	8.80	306	9.44	306	9.52	306	9.31
	A0 S2 C1	330	8.67	326	8.19	326	9.41	326	9.38
	A0 S2 C2	328	8.80	306	9.44	306	9.52	306	9.31
	A0 S2 C3	330	8.67	326	8.19	326	9.41	326	9.38
	A0 S3 C0	317	8.39	286	8.58	286	8.76	286	8.13
	A0 S3 C1	315	8.71	299	8.47	299	9.11	299	9.09
	A0 S3 C2	317	8.39	286	8.58	286	8.76	286	8.13
	A0 S3 C3	315	8.71	299	8.47	299	9.11	299	9.09
	A1 S0 C0	352	10.38	346	10.62	346	10.81	346	10.91
	A1 S0 C1	353	10.66	348	11.38	348	11.47	348	11.13
	A1 S0 C2	352	10.38	346	10.62	346	10.81	346	10.91
	A1 S0 C3	353	10.66	348	11.38	348	11.47	348	11.13
	A1 S1 C0	318	9.04	288	8.73	288	8.63	288	8.56
	A1 S1 C1	314	8.99	293	9.27	293	9.08	293	9.28
	A1 S1 C2	318	9.04	288	8.73	288	8.63	288	8.56

Tabelle A.18: Ergebnisse der Controllersynthese für den Chipkartenleser

Controllertyp	Optionen	T0		T1		T2		T3	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	A1 S1 C3	314	8.99	293	9.27	293	9.08	293	9.28
	A1 S2 C0	329	8.15	307	9.15	307	9.13	307	9.46
	A1 S2 C1	331	8.15	327	8.86	327	8.78	327	8.67
	A1 S2 C2	329	8.15	307	9.15	307	9.13	307	9.46
	A1 S2 C3	331	8.15	327	8.86	327	8.78	327	8.67
	A1 S3 C0	318	9.04	288	8.27	288	8.39	288	8.81
	A1 S3 C1	314	8.16	293	9.18	293	8.99	293	9.37
	A1 S3 C2	318	9.04	288	8.27	288	8.39	288	8.81
	A1 S3 C3	314	8.16	293	9.18	293	8.99	293	9.37
Single Process Distributed	A0 S0 C0	351	9.58	343	9.40	343	9.08	343	8.81
	A0 S0 C1	353	9.98	347	8.21	347	8.41	347	8.81
	A0 S0 C2	351	9.58	343	9.40	343	9.08	343	8.81
	A0 S0 C3	353	9.98	347	8.21	347	8.41	347	8.81
	A0 S1 C0	318	7.33	289	9.19	289	9.09	289	9.18
	A0 S1 C1	318	7.15	295	8.85	295	9.10	295	8.44
	A0 S1 C2	318	7.33	289	9.19	289	9.09	289	9.18
	A0 S1 C3	318	7.15	295	8.85	295	9.10	295	8.44
	A0 S2 C0	330	8.88	309	9.65	309	9.32	309	8.37
	A0 S2 C1	330	8.41	326	9.54	326	9.18	326	9.48
	A0 S2 C2	330	8.88	309	9.65	309	9.32	309	8.37
	A0 S2 C3	330	8.41	326	9.54	326	9.18	326	9.48
	A0 S3 C0	318	7.33	289	8.45	289	7.75	289	8.02
	A0 S3 C1	318	7.92	295	8.79	295	8.71	295	8.76
	A0 S3 C2	318	7.33	289	8.45	289	7.75	289	8.02
	A0 S3 C3	318	7.92	295	8.79	295	8.71	295	8.76
	A1 S0 C0	350	10.59	340	12.01	340	12.28	340	12.52
	A1 S0 C1	350	9.62	348	10.59	348	11.57	348	10.97
	A1 S0 C2	350	10.59	340	12.01	340	12.28	340	12.52
	A1 S0 C3	350	9.62	348	10.59	348	11.57	348	10.97
	A1 S1 C0	320	7.97	289	9.55	289	9.33	289	8.84
	A1 S1 C1	319	7.56	298	9.37	298	9.08	298	9.24
	A1 S1 C2	320	7.97	289	9.55	289	9.33	289	8.84
	A1 S1 C3	319	7.56	298	8.91	298	9.26	298	9.48
	A1 S2 C0	331	8.94	308	9.43	308	9.03	308	9.40
	A1 S2 C1	330	8.31	325	8.78	325	8.92	325	8.87
	A1 S2 C2	331	8.94	308	9.43	308	9.03	308	9.40
	A1 S2 C3	330	8.31	325	8.78	325	8.92	325	8.87
	A1 S3 C0	320	7.97	289	9.26	289	9.19	289	9.03
	A1 S3 C1	319	7.55	298	8.75	298	8.98	298	8.82
	A1 S3 C2	320	7.97	289	9.26	289	9.19	289	9.03
	A1 S3 C3	319	7.55	298	8.75	298	8.98	298	8.82
Single Process Partitioned	A0 S0 C0	395	6.37	356	10.24	356	9.69	356	9.96
	A0 S0 C1	395	6.69	365	10.29	365	10.25	365	10.36
	A0 S0 C2	395	6.37	356	10.24	356	9.69	356	9.96
	A0 S0 C3	395	6.69	365	10.29	365	10.25	365	10.36
	A0 S1 C0	367	5.85	293	8.49	293	8.83	293	8.64
	A0 S1 C1	367	4.89	314	7.87	314	8.46	314	9.98
	A0 S1 C2	367	5.85	293	8.49	293	8.83	293	8.64
	A0 S1 C3	367	4.89	314	7.87	314	8.46	314	9.98
	A0 S2 C0	366	5.59	307	9.82	307	9.21	307	9.46
	A0 S2 C1	363	5.90	324	8.74	324	8.15	324	8.14
	A0 S2 C2	366	5.59	307	9.82	307	9.21	307	9.46

Tabelle A.18: Ergebnisse der Controllersynthese für den Chipkartenleser

Controllertyp	Optionen	T0		T1		T2		T3	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	A0 S2 C3	363	5.90	324	8.74	324	8.15	324	8.14
	A0 S3 C0	367	5.85	293	8.49	293	8.83	293	8.64
	A0 S3 C1	367	4.89	314	10.01	314	9.57	314	10.29
	A0 S3 C2	367	5.85	293	8.49	293	8.83	293	8.64
	A0 S3 C3	367	4.89	314	10.01	314	9.57	314	10.29
	A1 S0 C0	394	6.42	351	9.79	351	9.73	351	10.13
	A1 S0 C1	392	7.12	365	8.73	365	9.82	365	9.02
	A1 S0 C2	394	6.42	351	9.79	351	9.73	351	10.13
	A1 S0 C3	392	7.12	365	8.73	365	9.82	365	9.02
	A1 S1 C0	365	5.37	295	8.20	295	8.68	295	8.29
	A1 S1 C1	367	5.02	314	9.49	314	8.88	314	9.74
	A1 S1 C2	365	5.37	295	8.20	295	8.68	295	8.29
	A1 S1 C3	367	5.02	314	9.49	314	8.88	314	9.74
	A1 S2 C0	369	6.33	308	9.39	308	9.18	308	9.39
	A1 S2 C1	366	5.80	322	8.96	322	8.29	322	9.32
	A1 S2 C2	369	6.33	308	9.39	308	9.18	308	9.39
	A1 S2 C3	366	5.80	322	8.96	322	8.29	322	9.32
	A1 S3 C0	365	5.37	295	8.20	295	8.68	295	8.29
	A1 S3 C1	367	5.02	314	10.14	314	10.07	314	10.11
	A1 S3 C2	365	5.37	295	8.20	295	8.68	295	8.29
	A1 S3 C3	367	5.02	314	10.17	314	9.62	314	8.28
Split Process Automatic	A0 S0 C0	382	9.64	391	8.83	391	8.25	391	8.68
	A0 S0 C1	387	9.63	384	8.66	384	8.32	384	9.29
	A0 S0 C2	382	9.64	391	8.83	391	8.25	391	8.68
	A0 S0 C3	387	9.63	384	8.66	384	8.32	384	9.29
	A0 S1 C0	338	8.62	304	8.62	304	8.64	304	8.29
	A0 S1 C1	341	8.65	308	10.35	308	9.86	308	10.29
	A0 S1 C2	338	8.62	304	8.62	304	8.64	304	8.29
	A0 S1 C3	341	8.65	308	10.35	308	9.86	308	10.29
	A0 S2 C0	357	8.60	318	8.75	318	9.39	318	9.00
	A0 S2 C1	358	8.46	334	8.34	334	8.69	334	9.00
	A0 S2 C2	357	8.60	318	8.75	318	9.39	318	9.00
	A0 S2 C3	358	8.46	334	8.34	334	8.69	334	9.00
	A0 S3 C0	339	7.91	305	7.98	305	8.39	305	8.12
	A0 S3 C1	343	8.79	312	9.59	312	9.74	312	9.92
	A0 S3 C2	339	7.91	305	7.98	305	8.39	305	8.12
	A0 S3 C3	343	8.79	312	9.59	312	9.74	312	9.92
	A1 S0 C0	387	9.13	387	10.57	387	10.52	387	11.23
	A1 S0 C1	393	9.54	382	10.45	382	10.49	382	9.38
	A1 S0 C2	387	9.13	387	10.57	387	10.52	387	11.23
	A1 S0 C3	393	9.54	382	10.45	382	10.49	382	9.38
	A1 S1 C0	339	8.41	302	8.57	302	8.79	302	8.61
	A1 S1 C1	341	8.58	307	8.15	307	8.68	307	9.56
	A1 S1 C2	339	8.41	302	8.57	302	8.79	302	8.61
	A1 S1 C3	341	8.58	307	8.15	307	8.68	307	9.56
	A1 S2 C0	358	8.06	318	9.42	318	9.30	318	9.67
	A1 S2 C1	358	8.52	338	9.35	338	9.21	338	8.47
	A1 S2 C2	358	8.06	318	9.42	318	9.30	318	9.67
	A1 S2 C3	358	8.52	338	9.35	338	9.21	338	8.47
	A1 S3 C0	337	8.07	303	8.50	303	8.61	303	8.21
	A1 S3 C1	345	8.06	310	9.60	310	9.17	310	9.79
	A1 S3 C2	337	8.07	303	8.50	303	8.61	303	8.21

Tabelle A.18: Ergebnisse der Controllersynthese für den Chipkartenleser

Controllertyp	Optionen	T0		T1		T2		T3	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	A1 S3 C3	345	8.06	310	9.60	310	9.17	310	9.79
Split Process Distributed	A0 S0 C0	385	10.47	391	9.58	391	9.22	391	9.35
	A0 S0 C1	386	9.82	384	8.79	384	9.15	384	9.05
	A0 S0 C2	385	10.47	391	9.58	391	9.22	391	9.35
	A0 S0 C3	386	9.82	384	8.79	384	9.15	384	9.05
	A0 S1 C0	341	8.10	307	8.94	307	8.45	307	8.89
	A0 S1 C1	340	8.64	308	9.52	308	9.29	308	9.41
	A0 S1 C2	341	8.10	307	8.94	307	8.45	307	8.89
	A0 S1 C3	340	8.64	308	9.52	308	9.29	308	9.41
	A0 S2 C0	352	8.64	319	9.11	319	9.20	319	9.62
	A0 S2 C1	351	8.40	341	8.61	341	8.85	341	8.76
	A0 S2 C2	352	8.64	319	9.11	319	9.20	319	9.62
	A0 S2 C3	351	8.40	341	8.61	341	8.85	341	8.76
	A0 S3 C0	339	8.42	306	8.44	306	8.69	306	8.70
	A0 S3 C1	340	8.31	309	9.54	309	9.65	309	9.80
	A0 S3 C2	339	8.42	306	8.44	306	8.69	306	8.70
	A0 S3 C3	340	8.31	309	9.54	309	9.65	309	9.80
	A1 S0 C0	391	9.82	387	10.75	387	10.52	387	10.71
	A1 S0 C1	392	9.87	383	10.48	383	10.75	383	11.17
	A1 S0 C2	391	9.82	387	10.75	387	10.52	387	10.71
	A1 S0 C3	392	9.87	383	10.48	383	10.75	383	11.17
	A1 S1 C0	341	7.92	306	7.40	306	7.51	306	7.48
	A1 S1 C1	343	8.71	309	9.97	309	9.80	309	10.15
	A1 S1 C2	341	7.92	306	7.40	306	7.51	306	7.48
	A1 S1 C3	343	8.71	309	9.97	309	9.80	309	10.15
	A1 S2 C0	353	8.12	319	9.34	319	9.05	319	9.27
	A1 S2 C1	355	8.54	342	8.84	342	9.18	342	8.78
	A1 S2 C2	353	8.12	319	9.34	319	9.05	319	9.27
	A1 S2 C3	355	8.54	342	8.84	342	9.18	342	8.78
	A1 S3 C0	340	8.62	306	8.87	306	8.36	306	8.47
	A1 S3 C1	344	8.10	311	9.61	311	9.57	311	10.06
	A1 S3 C2	340	8.62	306	8.87	306	8.36	306	8.47
	A1 S3 C3	344	8.10	311	9.61	311	9.57	311	10.06
Split Process Partitioned	A0 S0 C0	439	5.43	440	7.34	440	7.41	440	6.79
	A0 S0 C1	442	5.96	438	7.25	438	6.99	438	7.39
	A0 S0 C2	439	5.43	440	7.34	440	7.41	440	6.79
	A0 S0 C3	442	5.96	438	7.25	438	6.99	438	7.39
	A0 S1 C0	375	5.78	303	8.93	303	8.90	303	9.17
	A0 S1 C1	382	5.71	310	9.78	310	9.55	310	9.44
	A0 S1 C2	375	5.78	303	8.93	303	8.90	303	9.17
	A0 S1 C3	382	5.71	310	9.78	310	9.55	310	9.44
	A0 S2 C0	391	6.29	322	9.24	322	9.52	322	9.70
	A0 S2 C1	386	6.79	347	8.95	347	8.52	347	9.03
	A0 S2 C2	391	6.29	322	9.24	322	9.52	322	9.70
	A0 S2 C3	386	6.79	347	8.95	347	8.52	347	9.03
	A0 S3 C0	379	5.40	301	8.85	301	8.07	301	8.88
	A0 S3 C1	379	5.79	311	9.79	311	9.19	311	9.38
	A0 S3 C2	379	5.40	301	8.85	301	8.07	301	8.88
	A0 S3 C3	379	5.79	311	9.79	311	9.19	311	9.38
	A1 S0 C0	438	7.27	434	9.63	434	10.03	434	10.19
	A1 S0 C1	437	7.47	436	9.28	436	9.52	436	9.89
	A1 S0 C2	438	7.27	434	9.63	434	10.03	434	10.19

Tabelle A.18: Ergebnisse der Controllersynthese für den Chipkartenleser

Controllertyp	Optionen	T0		T1		T2		T3	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	A1 S0 C3	437	7.47	436	9.28	436	9.52	436	9.89
	A1 S1 C0	373	6.09	301	8.68	301	9.06	301	8.84
	A1 S1 C1	380	5.25	311	9.12	311	9.04	311	9.27
	A1 S1 C2	373	6.09	301	8.68	301	9.06	301	8.84
	A1 S1 C3	380	5.25	311	9.12	311	9.04	311	9.27
	A1 S2 C0	391	6.03	323	9.10	323	9.51	323	8.90
	A1 S2 C1	387	6.52	342	8.80	342	9.07	342	8.76
	A1 S2 C2	391	6.03	323	9.10	323	9.51	323	8.90
	A1 S2 C3	387	6.52	342	8.80	342	9.07	342	8.76
	A1 S3 C0	375	5.49	300	8.86	300	9.35	300	9.36
	A1 S3 C1	377	5.40	313	9.63	313	9.01	313	9.72
	A1 S3 C2	375	5.49	300	8.86	300	9.35	300	9.36
	A1 S3 C3	377	5.40	313	9.63	313	9.01	313	9.72
Multi Process Automatic	A0 S0 C0	386	10.60	396	9.11	396	9.27	396	9.39
	A0 S0 C1	383	10.36	387	8.31	387	8.39	387	9.57
	A0 S0 C2	386	10.60	396	9.11	396	9.27	396	9.39
	A0 S0 C3	383	10.36	387	8.31	387	8.39	387	9.57
	A0 S1 C0	341	8.48	306	7.91	306	8.00	306	8.09
	A0 S1 C1	341	8.86	310	9.00	310	9.31	310	8.68
	A0 S1 C2	341	8.48	306	7.91	306	8.00	306	8.09
	A0 S1 C3	341	8.86	310	9.00	310	9.31	310	8.68
	A0 S2 C0	354	8.57	317	9.52	317	9.58	317	9.29
	A0 S2 C1	358	8.60	333	9.40	333	9.67	333	9.28
	A0 S2 C2	354	8.57	317	9.52	317	9.58	317	9.29
	A0 S2 C3	358	8.60	333	9.40	333	9.67	333	9.28
	A0 S3 C0	342	8.54	307	8.56	307	8.38	307	8.97
	A0 S3 C1	346	7.85	312	9.06	312	9.19	312	9.95
	A0 S3 C2	342	8.54	307	8.56	307	8.38	307	8.97
	A0 S3 C3	346	7.85	312	9.06	312	9.19	312	9.95
	A1 S0 C0	387	10.71	392	10.53	392	11.82	392	11.66
	A1 S0 C1	387	10.78	384	11.74	384	11.30	384	11.49
	A1 S0 C2	387	10.71	392	10.53	392	11.82	392	11.66
	A1 S0 C3	387	10.78	384	11.74	384	11.30	384	11.49
	A1 S1 C0	339	8.84	306	8.22	306	8.74	306	8.74
	A1 S1 C1	343	8.20	309	9.90	309	9.56	309	9.66
	A1 S1 C2	339	8.84	306	8.22	306	8.74	306	8.74
	A1 S1 C3	343	8.20	309	9.90	309	9.56	309	9.66
	A1 S2 C0	356	8.08	318	9.00	318	8.94	318	9.23
	A1 S2 C1	363	7.79	337	8.20	337	8.98	337	8.36
	A1 S2 C2	356	8.08	318	9.00	318	8.94	318	9.23
	A1 S2 C3	363	7.79	337	8.20	337	8.98	337	8.36
	A1 S3 C0	340	9.09	307	8.64	307	8.35	307	8.46
	A1 S3 C1	347	8.78	312	9.91	312	10.32	312	9.80
	A1 S3 C2	340	9.09	307	8.64	307	8.35	307	8.46
	A1 S3 C3	347	8.78	312	9.91	312	10.32	312	9.80
Multi Process Distributed	A0 S0 C0	386	9.78	388	9.82	388	9.47	388	9.18
	A0 S0 C1	385	10.42	388	9.41	388	9.33	388	9.35
	A0 S0 C2	386	9.78	388	9.82	388	9.47	388	9.18
	A0 S0 C3	385	10.42	388	9.41	388	9.33	388	9.35
	A0 S1 C0	340	8.44	308	8.44	308	8.23	308	8.07
	A0 S1 C1	339	7.94	308	9.74	308	9.24	308	9.86
	A0 S1 C2	340	8.44	308	8.44	308	8.23	308	8.07

Tabelle A.18: Ergebnisse der Controllersynthese für den Chipkartenleser

Controllertyp	Optionen	T0		T1		T2		T3	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	A0 S1 C3	339	7.94	308	9.74	308	9.24	308	9.86
	A0 S2 C0	349	8.90	321	9.34	321	9.73	321	8.79
	A0 S2 C1	350	8.42	339	8.70	339	9.02	339	9.17
	A0 S2 C2	349	8.90	321	9.34	321	9.73	321	8.79
	A0 S2 C3	350	8.42	339	8.70	339	9.02	339	9.17
	A0 S3 C0	343	8.06	305	7.90	305	7.97	305	8.37
	A0 S3 C1	340	8.47	309	9.35	309	9.36	309	9.49
	A0 S3 C2	343	8.06	305	7.90	305	7.97	305	8.37
	A0 S3 C3	340	8.47	309	9.35	309	9.36	309	9.49
	A1 S0 C0	392	10.25	388	10.48	388	11.46	388	11.31
	A1 S0 C1	389	10.29	384	10.87	384	11.21	384	11.71
	A1 S0 C2	392	10.25	388	10.48	388	11.46	388	11.31
	A1 S0 C3	389	10.29	384	10.87	384	11.21	384	11.71
	A1 S1 C0	337	8.62	307	8.48	307	8.47	307	8.47
	A1 S1 C1	337	8.37	307	9.36	307	9.35	307	10.20
	A1 S1 C2	337	8.62	307	8.48	307	8.47	307	8.47
	A1 S1 C3	337	8.37	307	9.36	307	9.35	307	10.20
	A1 S2 C0	349	8.21	321	9.31	321	9.11	321	9.60
	A1 S2 C1	350	8.57	340	7.98	340	7.81	340	8.74
	A1 S2 C2	349	8.21	321	9.31	321	9.11	321	9.60
	A1 S2 C3	350	8.57	340	7.98	340	7.81	340	8.74
	A1 S3 C0	343	8.25	305	8.26	305	8.29	305	8.54
	A1 S3 C1	340	8.70	309	9.56	309	9.67	309	9.93
	A1 S3 C2	343	8.25	305	8.26	305	8.29	305	8.54
	A1 S3 C3	340	8.70	309	9.56	309	9.67	309	9.93
Multi Process Partitioned	A0 S0 C0	439	6.12	430	6.87	430	6.87	430	7.26
	A0 S0 C1	442	5.61	439	6.44	439	6.19	439	6.77
	A0 S0 C2	439	6.12	430	6.87	430	6.87	430	7.26
	A0 S0 C3	442	5.61	439	6.44	439	6.19	439	6.77
	A0 S1 C0	378	4.88	307	9.43	307	9.13	307	9.04
	A0 S1 C1	375	5.49	313	8.93	313	8.18	313	9.76
	A0 S1 C2	378	4.88	307	9.43	307	9.13	307	9.04
	A0 S1 C3	375	5.49	313	8.93	313	8.18	313	9.76
	A0 S2 C0	387	5.91	321	9.13	321	9.15	321	9.32
	A0 S2 C1	384	6.47	345	8.60	345	8.79	345	8.43
	A0 S2 C2	387	5.91	321	9.13	321	9.15	321	9.32
	A0 S2 C3	384	6.47	345	8.60	345	8.79	345	8.43
	A0 S3 C0	380	5.53	304	9.08	304	8.16	304	8.99
	A0 S3 C1	378	6.15	314	9.15	314	9.16	314	8.89
	A0 S3 C2	380	5.53	304	9.08	304	8.16	304	8.99
	A0 S3 C3	378	6.15	314	9.15	314	9.16	314	8.89
	A1 S0 C0	437	7.18	425	10.07	425	9.76	425	9.81
	A1 S0 C1	437	6.84	438	8.33	438	9.10	438	9.38
	A1 S0 C2	437	7.18	425	10.07	425	9.76	425	9.81
	A1 S0 C3	437	6.84	438	8.33	438	9.10	438	9.38
	A1 S1 C0	378	6.24	305	8.80	305	8.95	305	8.86
	A1 S1 C1	375	5.77	320	9.09	320	9.39	320	9.32
	A1 S1 C2	378	6.24	305	8.80	305	8.95	305	8.86
	A1 S1 C3	375	5.77	320	9.09	320	9.39	320	9.32
	A1 S2 C0	388	6.52	321	9.72	321	9.58	321	9.87
	A1 S2 C1	385	5.96	344	8.88	344	8.67	344	9.02
	A1 S2 C2	388	6.52	321	9.72	321	9.58	321	9.87

Tabelle A.18: Ergebnisse der Controllersynthese für den Chipkartenleser

Controllertyp	Optionen	T0		T1		T2		T3	
		CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
	A1 S2 C3	385	5.96	344	8.88	344	8.67	344	9.02
	A1 S3 C0	381	5.66	303	8.89	303	8.77	303	9.06
	A1 S3 C1	380	5.56	321	9.63	321	9.67	321	9.91
	A1 S3 C2	381	5.66	303	8.89	303	8.77	303	9.06
	A1 S3 C3	380	5.56	321	9.63	321	9.67	321	9.91

Tabelle A.18: Ergebnisse der Controllersynthese für den Chipkartenleser

A.5 Entwurf eines Kryptographie-Datenpfades

Die Synthese einer Stufe eines regulär aufgebauten Kryptographie-Datenpfades für die IDEA-Verschlüsselung [Ascom98] wird in diesem Abschnitt in ihren RTL- und High-Level-Ergebnissen zusammengefaßt. Bei beiden Synthesemethoden war die Variation der Flächenvorgabe, der Strukturoptimierungen, der Übersetzungsoptionen und der Taktvorgaben möglich, deren Belegungen in den jeweiligen Ergebnistabellen entsprechend Tabelle A.19 bezeichnet sind. Als Zielhardware wurde bei beiden Synthesemethoden ein Xilinx 4052XL-3 FPGA verwendet.

Einstellungscode	Wert	Bedeutung
T	0	keine Taktvorgabe
	1	Taktvorgabe 1MHz
	2	Taktvorgabe 2MHz
	3	Taktvorgabe 3MHz
A	0	keine Flächenbegrenzung
	1	minimale Fläche (`set_max_area 0`)
S	0	keine Strukturierungsoption
	1	Strukturierungs-Optimierung mit Boolescher Option
	2	Strukturierungs-Optimierung mit Timing-Option
	3	Strukturierungs-Optimierung mit Boolescher und Timing-Option
C	0	`compile` ohne Zusatzoptionen
	1	`compile` mit `incremental_map`
	2	`compile` mit `prioritize_min_paths`
	3	`compile` mit `incremental_map` und `prioritize_min_paths`

Tabelle A.19: RTL-Syntheseeinstellungen für die IDEA-Stufe

A.5.1 RTL-Synthese

Die Synthese des RTL-Modells erfolgte mit dem Design-Compiler 1999.05. An diese schloß sich die Plazierung und Verdrahtung des FPGAs mit Xilinx M1.5i an, woraus die Ergebnisse nach Tabelle A.20 resultierten.

Optionen	C0		C1		C2		C3	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
T0 A0 S0	1648	1.31	1571	1.38	1648	1.31	1571	1.38
T0 A0 S1	1628	1.11	1553	1.33	1628	1.11	1553	1.33
T0 A0 S2	1609	1.24	1544	1.31	1609	1.24	1544	1.31
T0 A0 S3	1628	1.11	1553	1.33	1628	1.11	1553	1.33
T0 A1 S0	1691	1.24	1575	1.28	1691	1.24	1575	1.28
T0 A1 S1	1653	1.31	1563	1.35	1643	1.29	1563	1.35
T0 A1 S2	1623	1.22	1556	1.27	1624	1.25	1556	1.27

Tabelle A.20: Ergebnisse der RTL-Synthese der IDEA-Stufe

Optionen	C0		C1		C2		C3	
	CLBs	MHz	CLBs	MHz	CLBs	MHz	CLBs	MHz
T0 A1 S3	1651	1.27	1563	1.35	1644	1.19	1563	1.35
T1 A0 S0	1679	1.50	1571	1.36	1679	1.50	1571	1.36
T1 A0 S1	1651	1.47	1553	1.38	1651	1.47	1553	1.38
T1 A0 S2	1640	1.41	1544	1.46	1640	1.41	1544	1.46
T1 A0 S3	1651	1.47	1553	1.38	1651	1.47	1553	1.38
T1 A1 S0	1721	1.25	1582	1.45	1719	1.39	1582	1.45
T1 A1 S1	1686	1.26	1562	1.50	1686	1.26	1562	1.50
T1 A1 S2	1656	1.35	1552	1.50	1661	1.36	1556	1.40
T1 A1 S3	1687	1.35	1562	1.50	1687	1.35	1562	1.50
T2 A0 S0	1723	0.94	1571	1.05	1723	0.94	1571	1.05
T2 A0 S1	1795	0.82	1553	1.12	1795	0.82	1553	1.12
T2 A0 S2	1702	1.01	1544	1.10	1702	1.01	1544	1.10
T2 A0 S3	1795	0.82	1553	1.12	1795	0.82	1553	1.12
T2 A1 S0	1778	0.78	1582	0.82	1777	0.94	1582	0.82
T2 A1 S1	1794	0.78	1562	1.01	1794	0.64	1562	1.01
T2 A1 S2	1776	0.78	1556	0.90	1777	0.70	1548	1.05
T2 A1 S3	1770	0.79	1561	1.03	1811	0.75	1561	1.03

Tabelle A.20: Ergebnisse der RTL-Synthese der IDEA-Stufe

Der Einfluß der Taktvorgabe wurde in einer weiteren Versuchsreihe mit Vorgaben von 0,1 bis 2,3MHz in Schritten zu 500kHz untersucht. Hierbei wurden neben der minimalen Flächenvorgabe beide Strukturoptimierungen und keine `compile`-Optionen aktiviert (Tabelle A.21).

Vorgabe MHz	Ergebnis		Vorgabe MHz	Ergebnis		Vorgabe MHz	Ergebnis	
	CLBs	MHz		CLBs	MHz		CLBs	MHz
0.1	1684	1.36	0.9	1686	1.31	1.7	1772	0.86
0.2	1684	1.34	1.0	1684	1.26	1.8	1794	0.75
0.3	1684	1.33	1.1	1712	1.02	1.9	1791	0.84
0.4	1684	1.36	1.2	1750	1.07	2.0	1796	0.82
0.5	1684	1.38	1.3	1769	1.04	2.1	1791	0.80
0.6	1684	1.36	1.4	1743	1.13	2.2	1787	0.71
0.7	1684	1.37	1.5	1773	1.02	2.3	1781	0.71
0.8	1684	1.44	1.6	1775	1.07			

Tabelle A.21: Ergebnisse der taktgesteuerten RTL-Synthese der IDEA-Stufe

A.5.2 High-Level-Synthese

Die High-Level-Synthese der IDEA-Stufe wurde der Version 1999.05 des Behavior-Compilers und des Design-Compilers ausgeführt. Die Plazierung und Verdrahtung des FPGAs wurde mit Xilinx M1.5i vorgenommen. Die durch die Variationen der Syntheseoptionen erzielten Ergebnisse sind in Tabelle A.22 aufgeführt.

Optionen	A0				A1			
	CLBs	MHz	Takte	Zeit / [ms]	CLBs	MHz	Takte	Zeit / [ms]
T1 S0 C0	1616	1.90	0002	1.05	1602	1.78	0002	1.13
T1 S0 C1	1591	1.80	0002	1.11	1587	1.81	0002	1.11
T1 S0 C2	1616	1.90	0002	1.05	1600	1.71	0002	1.17
T1 S0 C3	1591	1.80	0002	1.11	1584	1.81	0002	1.10

Tabelle A.22: Ergebnisse der High-Level-Synthese der IDEA-Stufe

Optionen	A0				A1			
	CLBs	MHz	Takte	Zeit / [ms]	CLBs	MHz	Takte	Zeit / [ms]
T1 S1 C0	1573	1.82	0002	1.10	1556	1.95	0002	1.03
T1 S1 C1	1553	1.71	0002	1.17	1546	1.74	0002	1.15
T1 S1 C2	1573	1.82	0002	1.10	1559	1.74	0002	1.15
T1 S1 C3	1553	1.71	0002	1.17	1545	1.83	0002	1.09
T1 S2 C0	1574	1.98	0002	1.01	1560	1.72	0002	1.16
T1 S2 C1	1553	1.71	0002	1.17	1546	1.85	0002	1.08
T1 S2 C2	1574	1.98	0002	1.01	1560	1.59	0002	1.25
T1 S2 C3	1553	1.71	0002	1.17	1547	1.85	0002	1.08
T1 S3 C0	1573	1.82	0002	1.10	1558	1.78	0002	1.12
T1 S3 C1	1553	1.71	0002	1.17	1546	1.81	0002	1.11
T1 S3 C2	1573	1.82	0002	1.10	1558	1.84	0002	1.09
T1 S3 C3	1553	1.71	0002	1.17	1545	1.73	0002	1.15
T2 S0 C0	1006	2.33	0003	1.29	1002	2.54	0003	1.18
T2 S0 C1	0962	2.78	0003	1.08	0961	2.63	0003	1.14
T2 S0 C2	1006	2.33	0003	1.29	1000	2.28	0003	1.32
T2 S0 C3	0962	2.78	0003	1.08	0960	2.75	0003	1.09
T2 S1 C0	0957	2.56	0003	1.17	0957	2.61	0003	1.15
T2 S1 C1	0919	2.87	0003	1.05	0920	2.66	0003	1.13
T2 S1 C2	0957	2.56	0003	1.17	0957	2.88	0003	1.04
T2 S1 C3	0919	2.87	0003	1.05	0920	2.65	0003	1.13
T2 S2 C0	0954	2.65	0003	1.13	0960	2.47	0003	1.21
T2 S2 C1	0925	2.73	0003	1.10	0925	2.58	0003	1.16
T2 S2 C2	0954	2.65	0003	1.13	0960	2.71	0003	1.11
T2 S2 C3	0925	2.73	0003	1.10	0926	2.53	0003	1.18
T2 S3 C0	0957	2.56	0003	1.17	0955	2.74	0003	1.10
T2 S3 C1	0919	2.87	0003	1.05	0919	2.46	0003	1.22
T2 S3 C2	0957	2.56	0003	1.17	0957	2.62	0003	1.15
T2 S3 C3	0919	2.87	0003	1.05	0920	2.46	0003	1.22
T3 S0 C0	1159	3.40	0006	1.76	1166	3.11	0006	1.93
T3 S0 C1	1107	3.81	0006	1.58	1108	3.67	0006	1.63
T3 S0 C2	1159	3.40	0006	1.76	1167	3.38	0006	1.77
T3 S0 C3	1107	3.81	0006	1.58	1108	3.27	0006	1.83
T3 S1 C0	1029	3.34	0006	1.80	1043	3.71	0006	1.62
T3 S1 C1	0981	4.45	0006	1.35	0988	4.23	0006	1.42
T3 S1 C2	1029	3.34	0006	1.80	1043	3.42	0006	1.76
T3 S1 C3	0981	4.45	0006	1.35	0988	3.79	0006	1.59
T3 S2 C0	1018	3.98	0006	1.51	1029	3.95	0006	1.52
T3 S2 C1	0991	3.99	0006	1.50	0998	4.25	0006	1.41
T3 S2 C2	1018	3.98	0006	1.51	1029	4.08	0006	1.47
T3 S2 C3	0991	3.99	0006	1.50	0998	3.92	0006	1.53
T3 S3 C0	1025	3.87	0006	1.55	1038	3.29	0006	1.83
T3 S3 C1	0981	4.45	0006	1.35	0988	3.85	0006	1.56
T3 S3 C2	1025	3.87	0006	1.55	1039	4.08	0006	1.47
T3 S3 C3	0981	4.45	0006	1.35	0988	3.71	0006	1.62

Tabelle A.22: Ergebnisse der High-Level-Synthese der IDEA-Stufe

In einer zusätzlichen Versuchsreiche wurde der Einfluß der Taktvorgaben näher untersucht, wobei neben der Flächenminimierung beide Strukturoptimierungen und keine `compile`-Optionen aktiviert waren. Die Taktvorgabe wurde von 0,5 bis 19,5MHz in Schritten zu 500kHz variiert (Tabelle A.23).

Vorgabe / [MHz]	Ergebnis				Vorgabe / [MHz]	Ergebnis			
	CLBs	MHz	Takte	Zeit / [ms]		CLBs	MHz	Takte	Zeit / [ms]
0.5	1608	1.24	0002	1.62	10.5	0918	2.60	0019	7.30
1.0	1616	1.75	0002	1.14	11.0	1167	2.69	0022	8.18
1.5	1014	2.52	0003	1.19	11.5	1116	3.16	0022	6.96
2.0	1006	2.33	0003	1.29	12.0	1101	4.46	0022	4.93
2.5	1176	2.73	0004	1.47	12.5	1130	2.93	0022	7.51
3.0	1151	3.29	0006	1.82	13.0	1136	4.60	0023	5.00
3.5	0923	4.02	0007	1.74	13.5	1152	4.59	0023	5.01
4.0	0737	4.61	0013	2.82	14.0	1135	3.95	0023	5.82
4.5	0751	5.50	0013	2.36	14.5	1099	3.75	0029	7.74
5.0	0745	5.66	0013	2.30	15.0	1156	4.12	0029	7.04
5.5	0742	5.74	0013	2.26	15.5	1132	3.66	0029	7.92
6.0	0737	6.15	0013	2.11	16.0	1129	3.60	0029	8.06
6.5	0768	4.94	0013	2.63	16.5	1102	3.67	0030	8.18
7.0	0846	2.90	0013	4.48	17.0	1132	3.96	0030	7.57
7.5	1103	5.55	0017	3.06	17.5	1171	4.64	0030	6.46
8.0	0817	3.42	0018	5.27	18.0	1114	4.17	0033	7.92
8.5	0815	4.72	0019	4.03	18.5	1171	3.89	0033	8.47
9.0	0867	2.60	0019	7.30	19.0	1200	3.77	0033	8.76
9.5	0875	3.18	0019	5.98	19.5	1177	4.40	0033	7.49
10.0	0860	4.24	0019	4.48					

Tabelle A.23: Ergebnisse der taktgesteuerten HL-Synthese der IDEA-Stufe

Anhang B

In diesem Anhang werden die Verilog- und Protocol-Compiler-Modelle der im Hauptteil beschriebenen und ausgewerteten Entwürfe dokumentiert.

B.1 Bildschirmcontroller discount

B.1.1 RTL-Synthese

Dieser Abschnitt enthält die RTL-Modelle für den Bildschirmcontroller discount entsprechend [Golze99] (Listing B.1) und [Friedr98] (Listing B.2).

```
//-------------------------------------------------------------------  //000
//                                                                     //001
// DISCOUNT: Display-Controller                                        //002
//                                                                     //003
// Stellt einen 8-Bit-Speicher auf einem Video-Monitor monochrom dar   //004
// (Aufloesung 640x192, Bildfrequenz 50 Hz).                           //005
//                                                                     //006
// Das Modell ist in den Abschnitten 5.1-3 ausfuehrlich dokumentiert.  //007
//                                                                     //008
//-------------------------------------------------------------------  //009
                                                                       //010
                                // horizontale Parameter in Takten:    //011
`define H1    96                // linker Bildrand                     //012
`define H    640                // Bildpunkte                          //013
`define H2    88                // rechter Bildrand                    //014
`define H3    88                // horizontale Synchronisation         //015
`define Hline `H1+`H+`H2+`H3    // Zeilenlaenge                        //016
                                                                       //017
                                // vertikale Parameter in Zeilen:      //018
`define V1    58                // oberer Bildrand                     //019
`define V    192                // Bildbereich                         //020
`define V2    48                // unterer Bildrand                    //021
`define V3    16                // vertikale Synchronisation           //022
`define Vframe `V1+`V+`V2+`V3   // Bildaufbau in Zeilen                //023
                                                                       //024
// Definitionen                                                        //025
`define Asz   13                // Adressbreite - 1                    //026
`define Dsz    7                // Datenbreite  - 1                    //027
`define Bsz    7                // Byte-Breite - 1                     //028
`define Msz   `H*`V/(`Bsz+1)-1  // Speichergroesse - 1                 //029
                                                                       //030
//-------------------------------------------------------------------  //031
//                                                                     //032
// Modul discount                                                      //033
//                                                                     //034
// Display-Controller (Cathode-Ray-Tube-Controller)                    //035
//                                                                     //036
//-------------------------------------------------------------------  //037
                                                                       //038
module discount (ADDR, PIXEL, CSYNC, HSYNC, VSYNC, DATA, CLOCK, NRST);  //039
                                                                       //040
  // Ausgaenge                                                         //041
  output [`Asz:0] ADDR;   // Adresse fuer Bildspeicher                  //042
  output          PIXEL,  // Bildpunkt (0: dunkel, 1: hell)           //043
                  CSYNC,  // kombinierte Synchronisation              //044
                          //   (HSYNC und VSYNC)                      //045
                  HSYNC,  // horizontale (Zeilen-)Synchronisation     //046
                          //   (1-aktiv)                              //047
                  VSYNC;  // vertikale (Bild-)Synchronisation         //048
                          //   (1-aktiv)                              //049
                                                                       //050
  // Eingaenge                                                         //051
  input  [`Dsz:0] DATA;   // Daten aus dem Bildspeicher                 //052
  input           CLOCK,  // Takt (einphasig)                         //053
                  NRST;   // Initialisierung (0-aktiv)                //054
                                                                       //055
```

Listing B.1: Bildschirmcontroller discount in RTL-Verilog nach [Golze99]

```
  // Port-Typen                                                          //056
  wire   [`Asz:0] ADDR;                                                 //057
  wire   [`Dsz:0] DATA;                                                 //058
  wire            PIXEL,                                                //059
                  CSYNC,                                                //060
                  HSYNC,                                                //061
                  VSYNC,                                                //062
                  CLOCK,                                                //063
                  NRST;                                                 //064
                                                                        //065
  // interne Variablen                                                  //066
  wire            ADRIE,  // Freigabe: Bilddaten uebernehmen und       //067
                          //   Bildadresse erhoehen                     //068
                          //   (address increment enable)               //069
                  NXTAD,  // Bildadresse hochzaehlen                   //070
                  NXTLN;  // Zeile hochzaehlen                         //071
                                                                        //072
  //                                                                    //073
  // kombinierte Synchronisation                                        //074
  //                                                                    //075
  assign CSYNC = HSYNC ^ VSYNC;                                         //076
                                                                        //077
  //                                                                    //078
  // Instanzen                                                          //079
  //                                                                    //080
  hcount HCOUNT (NXTAD, NXTLN, HSYNC, CLOCK, NRST);                     //081
  vcount VCOUNT (ADRIE, VSYNC, NXTLN, CLOCK, NRST);                     //082
  memacc MEMACC (ADDR,  PIXEL, DATA,  ADRIE, NXTAD, CLOCK, NRST);       //083
                                                                        //084
endmodule  // discount                                                 //085
                                                                        //086
                                                                        //087
//---------------------------------------------------------------------  //088
//                                                                      //089
// Modul hcount                                                         //090
//                                                                      //091
// Timing der Zeilen; Steuersignale fuer Adressen, Bildpunkte,          //092
// horizontale Synchronisation und Timing des Bildes                    //093
//                                                                      //094
//---------------------------------------------------------------------  //095
//                                                                      //096
// Schnittstelle                                                        //097
//                                                                      //098
// NXTAD  im naechsten Takt Bildspeicherdaten uebernehmen,              //099
//        zur naechsten Bildspeicheradresse weiterschalten (1-aktiv)    //100
// NXTLN  im naechsten Takt Zeile hochzaehlen (1-aktiv)                 //101
// HSYNC  horizontale (Zeilen-)Synchronisation (1-aktiv)                //102
// CLOCK  Takt                                                          //103
// NRST   Initialisierung (0-aktiv)                                     //104
//                                                                      //105
//---------------------------------------------------------------------  //106
//                                                                      //107
// Timing und Arbeitsperiode von hcount:                                //108
//                                                                      //109
// `H1 Takte  linker Bildrand                                           //110
//            HSYNC=0, NXTAD=0, NTXLN=0                                 //111
// `H  Takte  Bildpunkte                                                //112
//            HSYNC=0, NXTLN=0, NXTAD wird alle 8 Takte gesetzt         //113
// `H2 Takte  rechter Bildrand                                          //114
//            HSYNC=0, NXTAD=0, NTXLN=0                                 //115
// `H3 Takte  horizontale Synchronisation                               //116
//            HSYNC=1, NXTAD=0, NXTLN wird im letzten Takt 1            //117
//                                                                      //118
//---------------------------------------------------------------------  //119
                                                                        //120
module hcount (NXTAD, NXTLN, HSYNC, CLOCK, NRST);                       //121
                                                                        //122
  // Ausgaenge                                                          //123
  output NXTAD,      // Bildadresse hochzaehlen                        //124
         NXTLN,      // Zeile hochzaehlen                              //125
         HSYNC;      // Zeilensynchronisation                          //126
                                                                        //127
  // Eingaenge                                                          //128
  input  CLOCK,      // Takt                                           //129
```

Listing B.1: Bildschirmcontroller discount in RTL-Verilog nach [Golze99]

```
        NRST;       // Initialisierung (0-aktiv)                        //130
                                                                        //131
  // Port-Typen                                                         //132
  reg    NXTAD,                                                        //133
         NXTLN,                                                        //134
         HSYNC;                                                        //135
  wire   CLOCK,                                                        //136
         NRST;                                                         //137
                                                                       //138
  // interne Variablen                                                 //139
  reg [9:0] HCNT;    // Taktzaehler innerhalb einer Zeile              //140
  reg [1:0] HPHASE;  // Phasen horizontal:                             //141
                     //   0 linker Bildrand                           //142
                     //   1 Bildpunkte                                //143
                     //   2 rechter Bildrand                          //144
                     //   3 horizontale Synchronisation               //145
                                                                       //146
  //                                                                  //147
  // Taktzaehler aktualisieren                                        //148
  // (Periode `Hline, Bereich 0..`Hline-1)                            //149
  //                                                                  //150
  always @(posedge CLOCK or negedge NRST) begin                       //151
    if (NRST == 0)          HCNT <= 0;           // Reset             //152
    else                                                              //153
      if (HCNT == `Hline-1) HCNT <= 0;           // neue Zeile        //154
      else                  HCNT <= HCNT+1;                           //155
  end                                                                 //156
                                                                      //157
  //                                                                  //158
  // Phasenzaehler aktualisieren                                      //159
  //                                                                  //160
  always @(posedge CLOCK or negedge NRST) begin                       //161
    if (NRST == 0)  HPHASE <= 0;        // Reset                    //162
    else                                                              //163
      case (HCNT)                                                     //164
        0:          HPHASE <= 0;        // linker Bildrand           //165
        `H1:        HPHASE <= HPHASE+1; // Bildpunkte                //166
        `H1+`H:     HPHASE <= HPHASE+1; // rechter Bildrand          //167
        `H1+`H+`H2: HPHASE <= HPHASE+1; // Bildsynchronisation       //168
      endcase                                                         //169
  end                                                                 //170
                                                                      //171
  //                                                                  //172
  // horizontale Synchronisation aktualisieren;                       //173
  // HSYNC wird ueber ein Register ausgegeben, um die Verzoegerung    //174
  // auf NXTAD von einem Takt zu kompensieren und so das Timing nach  //175
  // aussen einzuhalten                                               //176
  //                                                                  //177
  always @(posedge CLOCK or negedge NRST) begin                       //178
    if (NRST == 0)     HSYNC <= 0;   // Reset                       //179
    else                                                              //180
      if (HPHASE == 3) HSYNC <= 1;                                    //181
      else             HSYNC <= 0;                                    //182
  end                                                                 //183
                                                                      //184
  //                                                                  //185
  // NXTAD und NXTLN aktualisieren                                    //186
  //                                                                  //187
  always @(HCNT or HPHASE) begin                                      //188
    if ((HCNT[2:0] == 3'b001)                 // neues Byte           //189
      && (HPHASE == 1))              NXTAD = 1;                       //190
    else                             NXTAD = 0;                       //191
    if ((HCNT == 0) && (HPHASE == 3)) NXTLN = 1;                      //192
    else                             NXTLN = 0;                       //193
  end                                                                 //194
                                                                      //195
endmodule // hcount                                                   //196
                                                                      //197
                                                                      //198
//---------------------------------------------------------------------//199
//                                                                    //200
// Modul vcount                                                       //201
//                                                                    //202
// Timing des Bildes; Steuersignale fuer Adressen, Bildpunkte und     //203
```

Listing B.1: Bildschirmcontroller discount in RTL-Verilog nach [Golze99]

```
// vertikale Synchronisation                                              //204
//                                                                        //205
//------------------------------------------------------------------------//206
//                                                                        //207
// Schnittstelle                                                          //208
//                                                                        //209
//  ADRIE  Freigabe: Bilddaten uebernehmen und Bildadresse erhoehen       //210
//         (address increment enable) (1-aktiv)                           //211
//  VSYNC  vertikale (Bild-)Synchronisation (1-aktiv)                     //212
//  NXTLN  im naechsten Takt Zeile hochzaehlen (1-aktiv)                  //213
//  CLOCK  Takt                                                           //214
//  NRST   Initialisierung (0-aktiv)                                      //215
//                                                                        //216
//------------------------------------------------------------------------//217
//                                                                        //218
// Timing: die Arbeitsperiode von vcount dauert `Vframe Zeilen und        //219
// beginnt bei NXTLN=1 mit einer steigenden Taktflanke:                   //220
//                                                                        //221
// `V1 Zeilen  oberer Bildrand                                           //222
//             VSYNC=0, ADRIE=0                                           //223
// `V  Zeilen  Bildzeilen                                                //224
//             VSYNC=0, ADRIE=1                                           //225
// `V2 Zeilen  unterer Bildrand                                          //226
//             VSYNC=0, ADRIE=0                                           //227
// `V3 Zeilen  vertikale Synchronisation                                 //228
//             VSYNC=1, ADRIE=0                                           //229
//                                                                        //230
//------------------------------------------------------------------------//231
                                                                          //232
module vcount (ADRIE, VSYNC, NXTLN, CLOCK, NRST);                         //233
                                                                          //234
  // Ausgaenge                                                            //235
  output ADRIE,      // Freigabe: Bildadresse erhoehen                   //236
         VSYNC;      // Bildsynchronisation                              //237
                                                                          //238
  // Eingaenge                                                            //239
  input  NXTLN,      // Zeile hochzaehlen                                //240
         CLOCK,      // Takt                                             //241
         NRST;       // Initialisierung (0-aktiv)                        //242
                                                                          //243
  // Port-Typen                                                           //244
  reg    ADRIE,                                                         //245
         VSYNC;                                                         //246
  wire   NXTLN,                                                         //247
         CLOCK,                                                         //248
         NRST;                                                          //249
                                                                          //250
  // interne Variablen                                                    //251
  reg [8:0] VCNT;    // Zeilenzaehler                                    //252
  reg [1:0] VPHASE;  // Phasen vertikal:                                 //253
                     //   0 oberer Bildrand                             //254
                     //   1 Bildbereich                                 //255
                     //   2 unterer Bildrand                            //256
                     //   3 vertikale Synchronisation                   //257
                                                                          //258
  //                                                                     //259
  // Zeilenzaehler aktualisieren                                         //260
  // (Periode `Vframe, Bereich 0..`Vframe-1)                             //261
  //                                                                     //262
  always @(posedge CLOCK or negedge NRST) begin                         //263
    if (NRST == 0)             VCNT <= 0;         // Reset             //264
    else                                                                //265
      if (NXTLN == 1) begin                       // neue Zeile        //266
        if (VCNT == `Vframe-1) VCNT <= 0;         // neues Bild        //267
        else                   VCNT <= VCNT+1;                          //268
      end                                                               //269
  end                                                                   //270
                                                                        //271
  //                                                                    //272
  // Phasenzaehler aktualisieren                                        //273
  //                                                                    //274
  always @(posedge CLOCK or negedge NRST) begin                         //275
    if (NRST == 0)    VPHASE <= 0;        // Reset                     //276
    else                                                                //277
```

Listing B.1: Bildschirmcontroller discount in RTL-Verilog nach [Golze99]

```
      if (NXTLN == 1)                      // neue Zeile                 //278
        case (VCNT)                                                       //279
          0:          VPHASE <= 0;        // oberer Bildrand            //280
          `V1:        VPHASE <= VPHASE+1; // Bildbereich                //281
          `V1+`V:     VPHASE <= VPHASE+1; // unterer Bildrand           //282
          `V1+`V+`V2: VPHASE <= VPHASE+1; // vertikale Synchronisation  //283
        endcase                                                           //284
  end                                                                     //285
                                                                          //286
  //                                                                      //287
  // vertikale Synchronisation aktualisieren                              //288
  //                                                                      //289
  always @(posedge CLOCK or negedge NRST) begin                           //290
    if (NRST == 0)     VSYNC <= 0;        // Reset                      //291
    else                                                                  //292
      if (VPHASE == 3) VSYNC <= 1;                                        //293
      else             VSYNC <= 0;                                        //294
  end                                                                     //295
                                                                          //296
  //                                                                      //297
  // ADRIE aktualisieren                                                  //298
  //                                                                      //299
  always @(VPHASE) begin                                                  //300
    if (VPHASE == 1) ADRIE <= 1;                                          //301
    else             ADRIE <= 0;                                          //302
  end                                                                     //303
                                                                          //304
endmodule // vcount                                                       //305
                                                                          //306
                                                                          //307
//------------------------------------------------------------------------//308
//                                                                        //309
// Modul memacc                                                           //310
//                                                                        //311
// Adressen fuer den Bildspeicher generieren, Bilddaten als Bytes         //312
// lesen und als Punkte ausgeben                                          //313
//                                                                        //314
//------------------------------------------------------------------------//315
//                                                                        //316
// Schnittstelle                                                          //317
//                                                                        //318
//  ADDR   Adresse fuer Bildspeicher                                      //319
//  PIXEL  Bildpunkt (0: dunkel, 1: hell)                                 //320
//  DATA   Daten aus dem Bildspeicher                                     //321
//  ADRIE  Freigabe: Bilddaten uebernehmen und Bildadresse erhoehen       //322
//         (address increment enable)                                     //323
//  NXTAD  Daten uebernehmen, Bildadresse hochzaehlen                     //324
//  CLOCK  Takt                                                           //325
//  NRST   Initialisierung (0-aktiv)                                      //326
//                                                                        //327
//------------------------------------------------------------------------//328
//                                                                        //329
// Timing und Funktion:                                                   //330
//                                                                        //331
// Bei einer positiven Taktflanke und ADRIE=NXTAD=1 werden das            //332
// Bildpunkt-Schieberegister mit den momentanen Bildspeicherdaten         //333
// geladen und die Bildspeicheradresse erhoeht. Letztere liegt im         //334
// Bereich von 0 bis `Msz, worauf wieder 0 folgt.                         //335
// Ist ADRIE oder NXTAD 0, wird das Bildpunkt-Schieberegister             //336
// weitergeschoben und beim niedrigstwertigen Bit mit 0 gefuellt.         //337
//                                                                        //338
//------------------------------------------------------------------------//339
                                                                          //340
module memacc (ADDR, PIXEL, DATA, ADRIE, NXTAD, CLOCK, NRST);             //341
                                                                          //342
  // Ausgaenge                                                            //343
  output [`Asz:0] ADDR;   // Adresse fuer Bildspeicher                    //344
  output          PIXEL;  // Bildpunkt                                   //345
                                                                          //346
  // Eingaenge                                                            //347
  input  [`Dsz:0] DATA;   // Daten aus dem Bildspeicher                   //348
  input           ADRIE,  // Freigabe: Bildadresse erhoehen               //349
                  NXTAD,  // Bildadresse hochzaehlen                      //350
                  CLOCK,  // Takt                                         //351
```

Listing B.1: Bildschirmcontroller discount in RTL-Verilog nach [Golze99]

```
                  NRST;   // Initialisierung (0-aktiv)                  //352
                                                                        //353
  // Port-Typen                                                         //354
  reg    [`Asz:0] ADDR;                                                 //355
  wire   [`Dsz:0] DATA;                                                 //356
  wire            PIXEL,                                                //357
                  ADRIE,                                                //358
                  NXTAD,                                                //359
                  CLOCK,                                                //360
                  NRST;                                                 //361
                                                                        //362
  // interne Variablen                                                  //363
  reg    [`Dsz:0] PIXSR;  // Bildpunkt-Schieberegister                  //364
                                                                        //365
  //                                                                    //366
  // Bildpunkt aus Schieberegister abgreifen                            //367
  //                                                                    //368
  assign PIXEL = PIXSR[`Dsz];                                           //369
                                                                        //370
  //                                                                    //371
  // Bildspeicheradresse aktualisieren                                  //372
  //                                                                    //373
  always @(posedge CLOCK or negedge NRST) begin                         //374
    if (NRST == 0)                    ADDR <= 0;  // Reset            //375
    else                                                                //376
      if ((NXTAD == 1) && (ADRIE == 1)) begin     // neue Adresse       //377
        if (ADDR == `Msz) ADDR <= 0;              // Ende Bildspeicher  //378
        else              ADDR <= ADDR+1;                               //379
      end                                                               //380
  end                                                                   //381
                                                                        //382
  //                                                                    //383
  // Bildpunkt-Schieberegister aktualisieren                            //384
  //                                                                    //385
  always @(posedge CLOCK or negedge NRST) begin                         //386
    if (NRST == 0) PIXSR <= 0;                    // Reset              //387
    else                                                                //388
      if ((NXTAD == 1) && (ADRIE == 1))                                 //389
        PIXSR <= DATA;                            // neues Byte laden   //390
      else PIXSR <= {PIXSR[`Dsz-1:0], 1'b0};      // schieben           //391
  end                                                                   //392
                                                                        //393
endmodule // memacc                                                     //394
```

Listing B.1: Bildschirmcontroller discount in RTL-Verilog nach [Golze99]

```
// ---------------------------------------------------------------------------
// 'Cathode-Ray-Tube-Controller'
// Bildschirmcontroller zur monochromen Ansteuerung eines Videomonitors
// mit Daten aus einem 8-Bit Datenspeicher.
// Aufloesung: 640x192 Punkte, 50 Hz Bildfrequenz
//
// Die Problemstellung ist in Kap.5 des Vorlesungsskriptes 'Einfuehrung in
// den VLSI-Entwurf' von Prof. Dr. Golze ausfuehrlich dokumentiert.
//
// Mixed-Mode-Beschreibung auf Register-Transfer-Ebene,
// fuer RTL-Synthese mit Synopsys HDL-Compiler entworfen.
// Der CRTC wurde weitgehend neu entworfen, einige Konstrukte an das
// DISCOUNT-Modell von Peter Blinzer angelehnt.
// Auf eine Modellierung des Resetsignals NRST wurde verzichtet, da der
// Pin GR (global reset) des Ziel-FPGAs (Xilinx XC3000-Serie) alle
// Register mit log. 0 initialisiert und so den CRTC in den Grundzustand
// bringt.
// Zur Modellierung von NRST (und spaeteren Synthese des async. Resetsignals)
// muessten die Konstrukte 'always @(posedge CLK)' ersetzt werden durch:
// always @(posedge CLK or negedge NRST)
// if (NRST == 0) <Initialisierung der Register>
// else ...
//
// Autor: Arne Friedrichs, 12/97
// ---------------------------------------------------------------------------
```

Listing B.2: Bildschirmcontroller discount in RTL-Verilog nach [Friedr98]

```
module crtc (ADDR, PIXEL, CSYNC, HSYNC, VSYNC, DATA, CLKIN);
// ------------------------------------------------------------------------
// Top-Level-Modul des Bildschirmcontrollers
//
// CLKIN    Taktsignal
// Schnittstelle zum Bildspeicher:
// ADDR     Bildspeicheradresse
// DATA     Bildspeicherdaten
// Schnittstelle zum Monitor (bzw. Puffer und Mischer):
// PIXEL    Bildpunktinformation (hell/dunkel)
// HSYNC    horizontale Synchronisation der Bildzeilen
// VSYNC    vertikale Synchronisation des Bildes
// CSYNC    kombinierte Synchronisation durch XOR von HSYNC und VSYNC
// ------------------------------------------------------------------------

   input             CLKIN;   // Takteingang (14,3 MHz fuer 50Hz Bildfreq.)
   input      [7:0]  DATA;    // Daten aus Bildspeicher
   output    [13:0]  ADDR;    // Adresse im Bildspeicher
   output            PIXEL,   // Bildpunktinformation
                     CSYNC,   // composite Synchronisation
                     HSYNC,   // horizontale Synchronisation
                     VSYNC;   // vertikale Synchronisation

   wire      [9:0]   HCOUNT;  // Position in der Bildzeile
   wire      [8:0]   VCOUNT;  // aktuelle Zeile
   wire              LASTROW, // letzter Punkt der Zeile erreicht
                     LASTCOL, // letzte Zeile erreicht
                     VISIBLE; // naechste Position ist sichtbares Pixel

   // Modul-Instanzen

   hcounter       HCNT (CLKIN, HCOUNT, LASTROW);
   vcounter       VCNT (CLKIN, LASTROW, VCOUNT, LASTCOL);
   sync           SYNC (HCOUNT, VCOUNT, HSYNC, VSYNC, CSYNC);
   visibility     VISB (HCOUNT, VCOUNT, VISIBLE);
   memacc         MEMA (CLKIN, VISIBLE, LASTCOL, HCOUNT, DATA, ADDR, PIXEL);

endmodule
// ------------------------------------------------------------------------



module hcounter (CLK, HCOUNT, LASTROW);
// ------------------------------------------------------------------------
// Pixel- (Takt-) Zaehler innerhalb einer Bildschirmzeile.
// Eine Bildschirmzeile gliedert sich in 4 Bereiche :
//  96 Takte linker Bildrand       (  0 -  95)
// 640 Takte sichtbares Bild       ( 96 - 735)
//  88 Takte rechter Bildrand      (736 - 823)
//  88 Takte Zeilensynchronisation (824 - 911)
//
// CLK      Takt
// HCOUNT   Position in der Bildzeile
// LASTROW  log. 1, wenn der letzte Punkt der Zeile erreicht ist. Zeigt einen
//          Zeilenwechsel im naechsten Takt an.
// ------------------------------------------------------------------------

   input          CLK;        // Takt
   output   [9:0] HCOUNT;     // Position in der Bildzeile
   output         LASTROW;    // Zeilenwechsel im naechsten Takt

   reg      [9:0] HCOUNT;
   reg            LASTROW;


   // synopsys translate_off
   initial begin              // FPGA-Initialisierung durch GR
      HCOUNT <= 0;
   end
   // synopsys translate_on
```

Listing B.2: Bildschirmcontroller discount in RTL-Verilog nach [Friedr98]

```
   always @(posedge CLK) begin

      if (LASTROW)            // Taktzaehler
         HCOUNT <= 0;
      else
         HCOUNT <= HCOUNT + 1;
   end


   always @(HCOUNT) begin

      if (HCOUNT == 911)      // letzter Takt der Zeile ?
         LASTROW <= 1;
      else
         LASTROW <= 0;
   end

endmodule
// ---------------------------------------------------------------------------



module vcounter (CLK, ENABLE, VCOUNT, LASTCOL);
// ---------------------------------------------------------------------------
// Zeilenzaehler.
// Das Bild gliedert sich in 4 Bereiche:
//  58 Zeilen oberer Bildrand     (  0 -  57)
// 192 Zeilen sichtbares Bild     ( 58 - 249)
//  48 Zeilen unterer Bildrand    (250 - 297)
//  16 Zeilen Bildsynchronisation (298 - 313)
//
// CLK      Takt
// ENABLE   Count-/Counterreset-enable, nur bei Zeilenwechsel zaehlen
// VCOUNT   aktuelle Bildschirmzeile
// LASTCOL  letzte Bildschirmzeile
// ---------------------------------------------------------------------------

   input          CLK,        // Takteingang
                  ENABLE;     // count/reset enable, Zeilenwechsel
   output   [8:0] VCOUNT;     // aktuelle Zeile
   output         LASTCOL;    // letzte Zeile

   reg      [8:0] VCOUNT;
   reg            LASTCOL;


   // synopsys translate_off
   initial begin              // FPGA-Initialisierung durch GR
      VCOUNT <= 0;
   end
   // synopsys translate_on


   always @(posedge CLK) begin

      if (ENABLE)             // bei Zeilenwechsel Zaehler erhoehen,
         if (LASTCOL)         // bzw. zu 0 setzen
            VCOUNT <= 0;
         else
            VCOUNT <= VCOUNT + 1;
   end


   always @(VCOUNT) begin

      if (VCOUNT == 313)      // letzte Zeile ?
         LASTCOL <= 1;
      else
         LASTCOL <= 0;
   end

endmodule
// ---------------------------------------------------------------------------
```

Listing B.2: Bildschirmcontroller discount in RTL-Verilog nach [Friedr98]

```
module sync (HCOUNT, VCOUNT, HSYNC, VSYNC, CSYNC);
// ------------------------------------------------------------------------
// Synchronisationssignale erzeugen. (rein kombinatorisch)
//
// HCOUNT   horizontale Bildposition
// VCOUNT   vertikale Bildposition
// HSYNC    horizontale (Zeilen-) Synchronisation
// VSYNC    vertikale (Bild-) Synchronisation
// CSYNC    kombinierte Synchronisation (composite synchronization)
// ------------------------------------------------------------------------

   input    [9:0] HCOUNT;     // horizontale Position
   input    [8:0] VCOUNT;     // vertikale Position
   output         HSYNC,      // Zeilensynchronisation
                  VSYNC,      // Bildsynchronisation
                  CSYNC;      // kombinierte Synchronisation

   reg            HSYNC;
   reg            VSYNC;


   always @(HCOUNT)
      if (HCOUNT < 824)       // Zeilensynchronisation bei HCOUNT >= 824
         HSYNC <= 0;
      else
         HSYNC <= 1;


   always @(VCOUNT)           // Bildsynchronisation bei VCOUNT >= 298
      if (VCOUNT < 298)
         VSYNC <= 0;
      else
         VSYNC <= 1;


   assign CSYNC = HSYNC ^ VSYNC; // composite synchr. mittels XOR

endmodule
// ------------------------------------------------------------------------



module visibility (HCOUNT, VCOUNT, VISIBLE);
// ------------------------------------------------------------------------
// Ermitteln, ob die aktuelle Bildposition im sichtbaren Bereich liegt.
// Zur Steuerung, ob Bildspeicherdaten verarbeitet und ausgegeben werden
// muessen. (rein kombinatorische Logik)
//
// HCOUNT   horizontale Bildposition
// VCOUNT   vertikale   Bildposition
// VISIBLE  naechster (!) Bildpunkt ist sichtbar.
// ------------------------------------------------------------------------

   input    [9:0] HCOUNT;     // horizontale Position
   input    [8:0] VCOUNT;     // vertikale Position
   output         VISIBLE;    // NAECHSTER (!) Punkt ist sichtbar

   reg            VISIBLE;


   always @(HCOUNT or VCOUNT) begin

      if ((HCOUNT >= 95)      // Position im sichtbaren Bereich ?
       && (HCOUNT < 735)      // Zeilenposition um 1 verschoben, da
       && (VCOUNT >= 58)      // angezeigt wird, ob der naechste Punkt
       && (VCOUNT < 250))     // sichtbar ist.
         VISIBLE <= 1;
      else
         VISIBLE <= 0;
   end
```

Listing B.2: Bildschirmcontroller discount in RTL-Verilog nach [Friedr98]

```
endmodule
// ---------------------------------------------------------------------------



module memacc (CLK, VISIBLE, LASTCOL, HCOUNT, DATA, ADDR, PIXEL);
// ---------------------------------------------------------------------------
// Ansteuerung des Bildspeichers und Ausgabe der Pixelinformation.
// Der Bildspeicher wird byteweise adressiert und ausgelesen und
// bitweise als Punkt ausgegeben
//
// CLK      Takt
// VISIBLE  naechster Punkt liegt im sichtbaren Bildbereich
// LASTCOL  letzte Bildschirmzeile wird dargestellt
// HCOUNT   horizontale Bildposition
// DATA     Bildspeicherdaten
// ADDR     Bildspeicheradresse
// PIXEL    Pixelinformation (hell/dunkel)
// ---------------------------------------------------------------------------

   input             CLK,     // Takt
                     VISIBLE, // naechster Punkt sichtbar ?
                     LASTCOL; // letzte Bildzeile
   input    [9:0]    HCOUNT;  // Position in der Zeile
   input    [7:0]    DATA;    // Bildspeicherdaten
   output   [13:0]   ADDR;    // Bildspeicheradresse
   output            PIXEL;   // Pixelinformation

   reg      [13:0]   ADDR;
   reg      [7:0]    PIXELSR; // Schieberegister zur Aufnahme eines Bytes
                              // aus dem Bildspeicher

   // synopsys translate_off
   initial begin              // FPGA-Initialisierung durch GR
      ADDR <= 0;
      PIXELSR <= 0;
   end
   // synopsys translate_on


   assign PIXEL = PIXELSR[7]; // MSB des Schieberegisters enthaelt die
                              // aktuelle Pixelinformation

   always @(posedge CLK) begin

      // Falls der naechste Punkt sichtbar ist und HCOUNT+1 durch 8
      // teilbar ist (der erste sichtbare Punkt liegt bei HCOUNT = 96,
      // 96 mod 8 = 0, und auch die Anzahl der sichtbaren Bildpunkte pro
      // Zeile ist ein Vielfaches von 8), muss das Schieberegister neu
      // mit Bildspeicherdaten geladen werden, ansonsten wird ein
      // Bit nach links geschoben, um die naechste Bildpunktinf. im MSB
      // zu haben.
      if (VISIBLE && (HCOUNT[2:0] == 3'b111))
         PIXELSR <= DATA;
      else
         PIXELSR[7:0] <= { PIXELSR[6:0], 1'b0 };

      // Einen Takt nach dem Laden des Schieberegisters wird die
      // Bildspeicheradresse erhoeht, so bleiben dem Speicher 7 Takte
      // zum Ausgeben der Bilddaten, was die Verwendung von langsameren
      // Speichern erlaubt.
      if (VISIBLE && (HCOUNT[2:0] == 3'b000))
         ADDR <= ADDR + 1;

      if (LASTCOL)            // Wenn die letzte Bildzeile erreicht ist,
         ADDR <= 0;           // wird die Bildspeicheradresse zurueckgesetzt

   end

endmodule
// ---------------------------------------------------------------------------
```

Listing B.2: Bildschirmcontroller discount in RTL-Verilog nach [Friedr98]

B.1.2 High-Level-Synthese

Listing B.3 zeigt das High-Level-Modell des discount nach [Friedr98].

```
// ------------------------------------------------------------------------
// 'Cathode-Ray-Tube-Controller'
// Bildschirmcontroller zur monochromen Ansteuerung eines Videomonitors
// mit Daten aus einem 8-Bit Datenspeicher.
// Aufloesung: 640x192 Punkte, 50 Hz Bildfrequenz
//
// Die Problemstellung ist in Kap.5 des Vorlesungsskriptes 'Einfuehrung in
// den VLSI-Entwurf' von Prof. Dr. Golze ausfuehrlich dokumentiert.
//
// Verhaltensbeschreibung des CRTC, fuer die High-Level-Synthese
// mit dem Synopsys Behavioral Compiler.
//
// (5.) Verhaltensbeschreibung, die durch Verwendung von Zaehlern als
// Zustandsvariablen innerhalb EINER unendlichen Schleife (in diesem
// Fall der always-Block) mit einem Taktzyklus (Zustand) auskommt.
// Durch das Fehlen von conditional-exits (in for-Schleifen implizit
// vorhanden) kann jede CRTC-Ausgabe in einem einzigen Taktzyklus erreicht
// werden, und ist somit im 'cycle_fixed'-Scheduling-Mode synthetisierbar.
// Zeitoptimiert !
//
// Fuer die Verhaltenssynthese wird das Reset-Signal NRST benoetigt,
// da der PIN GR (global reset) des Ziel-FPGAs (Xilinx XC3000-Serie)
// zwar alle Register mit log. 0 initialisiert, dadurch aber nicht
// sichergestellt ist, dass der Steuerungsautomat (control-finite-state-
// machine) in den Anfangszustand versetzt wird, da dieser nicht zwingend
// mit Null codiert wird.
// Auf eine explizite Modellierung des Resetsignal mittels entsprechenden
// 'if (NRST==0) disable...'-Anweisungen nach jedem '@(posedge CLKIN)' wurde
// der Uebersicht halber verzichtet, stattdessen wird NRST implizit durch
// 'set_behavioral_reset NRST -active low'
// 'set_behavioral_async_reset' vor dem Scheduling erzeugt.
//
// Autor: Arne Friedrichs, 12/97
// ------------------------------------------------------------------------



module crtc (ADDR, PIXEL, CSYNC, HSYNC, VSYNC, DATA, CLKIN, NRST);
// ------------------------------------------------------------------------
// Verhaltensbeschreibung des Bildschirmcontrollers
//
// CLKIN    Taktsignal
// NRST     Resetsignal
// Schnittstelle zum Bildspeicher:
// ADDR     Bildspeicheradresse
// DATA     Bildspeicherdaten
// Schnittstelle zum Monitor (bzw. Puffer und Mischer):
// PIXEL    Bildpunktinformation (hell/dunkel)
// HSYNC    horizontale Synchronisation der Bildzeilen
// VSYNC    vertikale Synchronisation des Bildes
// CSYNC    kombinierte Synchronisation durch XOR von HSYNC und VSYNC
// ------------------------------------------------------------------------

   input    [7:0]    DATA;    // Daten aus Bildspeicher
   input             CLKIN,   // Takt
                     NRST;    // Reset, active low; nicht modelliert !
   output   [13:0]   ADDR;    // Adresse fuer Bildspeicher
   output            PIXEL,   // Bildpunktinformation
                     CSYNC,   // kombinierte Synchronisation
                     HSYNC,   // horizontale Synchronisation
                     VSYNC;   // vertikale Synchronisation

   wire     [7:0]    DATA;
   wire              CLKIN;
   reg      [13:0]   ADR,     // Zwischenspeicher fuer Berechnung von ADDR
                     ADDR;
   reg               HSYNC,
                     VSYNC,
                     PIXEL;
```

Listing B.3: Bildschirmcontroller in High-Level-Verilog nach [Friedr98]

```
reg      [9:0]    HCOUNT;  // Taktzaehler innerhalb einer Bildzeile
   //   0 -  95,  96 Takte linker Bildrand
   //  96 - 735, 640 Takte sichtbare Bildzeile
   // 736 - 823,  88 Takte rechter Bildrand
   // 824 - 911,  88 Takte Zeilensynchronisation

reg      [8:0]    VCOUNT;  // Zeilenzaehler
   //   0 -  57,  58 Zeilen oberer Bildrand
   //  58 - 249, 192 Zeilen sichtbar
   // 250 - 297,  48 Zeilen unterer Bildrand
   // 298 - 313,  16 Zeilen Bildsynchronisation

reg      [7:0]    PIXELSR; // Schieberegister fuer Bildpunktinformation


// synopsys translate_off
initial begin              // FPGA-Initialisierung durch GR
   HCOUNT = 0;
   VCOUNT = 0;
   ADR = 0;
   PIXELSR = 0;
   ADDR <= 0;
   HSYNC <= 0;
   VSYNC <= 0;
   PIXEL <= 0;
end
// synopsys translate_on


// composite synchronisation
assign CSYNC = HSYNC ^ VSYNC;


always begin : MAIN

   // Zeilensynchronisation
     if (HCOUNT == 0)        HSYNC <= 0;
     else if (HCOUNT == 824) HSYNC <= 1;

   // Bildsynchronisation
     if (VCOUNT == 0)        VSYNC <= 0;
     else if (VCOUNT == 298) VSYNC <= 1;


   // sichtbarer Bildbereich
   if ((HCOUNT >= 96) && (HCOUNT < 736) &&
      (VCOUNT >= 58) && (VCOUNT < 250)) begin : I2

      // Falls HCOUNT durch 8 teilbar ist (der erste sichtbare Punkt
      // liegt bei HCOUNT = 96, 96 mod 8 = 0, und auch die Anzahl
      // der sichtbaren Bildpunkte pro Zeile ist ein Vielfaches von 8),
      // muss das Schieberegister neu mit Bildspeicherdaten geladen
      // werden, ansonsten wird ein Bit nach links geschoben,
      // um die naechste Bildpunktinf. im MSB zu haben.
      if (HCOUNT[2:0] == 3'b000) begin
         PIXELSR = DATA;
      end
      else begin
         PIXELSR[7:0] = { PIXELSR[6:0], 1'b0 };

         // Einen Takt nach dem Laden des Schieberegisters wird die
         // Bildspeicheradresse erhoeht, so bleiben dem Speicher 7 Takte
         // zum Ausgeben der Bilddaten, was die Verwendung von langsameren
         // Speichern erlaubt.
         if (HCOUNT[2:0] == 3'b001) begin
            ADDR <= ADR;      // neue Adresse uebernehmen
            ADR = ADR + 1'b1; // da die Berechnung auf Variablen, nicht
         end                  // auf Signalen beruht, darf sie mehrere
                              // Taktzyklen dauern (multicycle OP.)
      end

      // Bildpunktinformation
      //
```

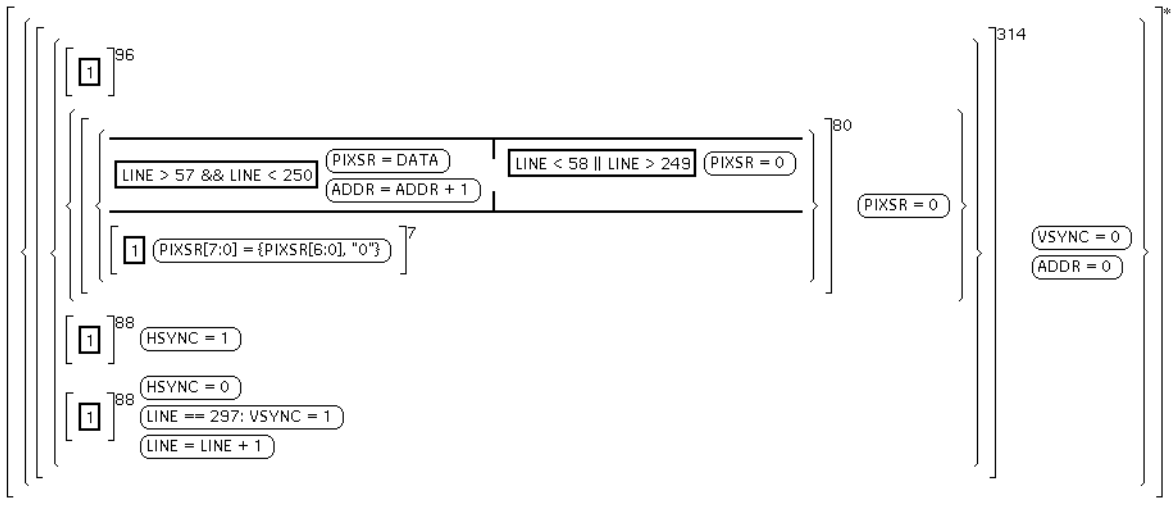
Listing B.3: Bildschirmcontroller in High-Level-Verilog nach [Friedr98]

```
         PIXEL <= PIXELSR[7];

      end
      else begin                 // nicht im sichtbaren Bereich
         PIXEL <= 0;
         if (VCOUNT == 0) begin  // Bildspeicheradresse ruecksetzen
            ADDR <= 0;
            ADR = 1;             // ADR = ADDR + 1;
         end
      end


      // Zaehlanweisungen 'nach hinten' Verschieben, um Datenabhaengigkeiten
      // zu brechen und so parallele Berechnung zu ermoeglichen

      // Taktzaehler innerhalb einer Zeile
      if (HCOUNT == 911) begin : I1
         HCOUNT = 0;

         // Zeilenzaehler
         if (VCOUNT == 313)
            VCOUNT = 0;
         else
            VCOUNT = VCOUNT + 1'b1;
      end
      else
         HCOUNT = HCOUNT + 1'b1;

      @(posedge CLKIN);

   end   // always
endmodule
// ---------------------------------------------------------------------------
```

Listing B.3: Bildschirmcontroller in High-Level-Verilog nach [Friedr98]

B.1.3 Controllersynthese

Bild B.1 zeigt das Protocol-Compiler-Modell des Bildschirmcontrollers.




```
Ports:                    Variablen:          Default Actions:
 ADDR[13:0], out           PIXSR[7:0]          CSYNC = HSYNC ^ VSYNC
 DATA[7:0], in             LINE[8:0]           PIXEL = PIXSR[7]
 PIXEL, out
 CSYNC, out
 HSYNC, out
 VSYNC, out
```

Bild B.1: Protocol-Compiler-Modell des Bildschirmcontrollers

B.2 Entwurf eines Digital-Audio-Receivers

B.2.1 RTL-Synthese

In diesem Abschnitt sind die RTL-Modelle für den Digital-Audio-Receiver gemäß [Blinze97] (Listing B.4) bzw. [Friedr98] (Listing B.5) zusammengefaßt.

```
//------------------------------------------------------------------------------
//
//  Modellbeschreibung:   Verilog-Modell eines Digital-Audio-Receivers
//
//  Projektbeschreibung:  Verilog-Modellierung eines Digital-Audio-Receivers
//                        Synthesegeeignet fuer Synopsys
//                        Entwurfs-Praktikum fuer FPGA-Schaltungen
//                        Technische Universitaet Braunschweig
//                        Abteilung Entwurf integrierter Schaltungen (E.I.S.)
//
//  Dateibezeichnung:     dar.v
//  Aktualisierungsdatum: 31. Maerz 1996
//  Implementiert von:    Peter Blinzer
//
//------------------------------------------------------------------------------
//
// Der nachfolgend in Verilog beschriebene Digital-Audio-Receiver dient
//  zur Umsetzung eines Digital-Audio-Signals in Steuerungsdaten fuer
//  einen 12-Bit Stereo D/A-Wandler des Typs AD8522 und ist fuer die
//  Implementierung in einem XILINX-FPGA der Serie XC3000 vorgesehen.
//  Es wird daher von einem impliziten 0-Reset aller Register bei
//  der Schaltungsinitialisierung ausgegangen, welcher in diesem Modell
//  mit "initial"-Bloecken ausgefuehrt wird.
//
// Das Digital-Audio-Signal besitzt das in der Norm IEC 958 fuer den
//  'Consumer Mode' spezifizierte Format. Bei der Umsetzung werden sowohl
//  die in jedem Datenblock vorhandenen Bits fuer Interpolationsanzeige
//  und Paritaetskontrolle verarbeitet, als auch der ueber 192 Frames aus
//  den Statusbits aufgebaute Statusblock ausgewertet. Das Vorhandensein
//  eines gueltigen 'IEC 958 Consumer Mode Signals' und der Kopiertstatus
//  des Signals werden ueber Ausgangssignale angezeigt. Bei ungueltigen
//  Signalen wird der D/A-Wandler stummgeschaltet.
//
// Die Schaltung besitzt folgende Eingaenge:
//   IECIN   Eingang fuer das Digital-Audio-Signal (biphasen-kodiert)
//   CLKIN   Eingang fuer den Arbeitstakt
//
// Die Schaltung besitzt folgende Ausgaenge:
//   DADAT   Daten fuer den seriellen D/A-Wandler
//   DACLK   Datentakt fuer den seriellen D/A-Wandler
//   DACSN   Selektionssignal fuer den seriellen D/A-Wandler
//   DALDN   Ladesignal fuer Umsetzungsregister des seriellen D/A-Wandlers
//   VALID   Anzeige eines gueltigen IEC-Signals (1 = gueltig)
//   COPYR   Anzeige des Kopierschutzstatus (1 = Kopiergeschuetzt)
//   COPYG   Anzeige der Kopiergeneration (0/1 = Generation 0/1)
//
//------------------------------------------------------------------------------
//
// Aufbau:
//   Der Digital-Audio Receiver (DAR) besteht aus den folgenden Komponenten:
//   1. BIPDEC (Biphasen-Decoder)
//        In diesem Modul werden die (nicht biphasen-kodierten)
//        Synchronisations-Praeambeln erkannt und die nachfolgenden Bits
//        gemaess dem Biphasen-Code dekodiert.
//   2. SHIFTIN (Empfangsschieberegister)
//        Dieses Register uebernimmt den seriellen Datenstrom aus dem
//        Biphasen-Decoder BIPDEC und gruppiert ihn zu Datenbloecken.
//   3. CONTROL (Kontroll-Logik)
//        In dieser Logik wird in Abhaengigkeit von Gueltigkeit, Inhalt
//        und Frame-Nummer des Datenblockes in SHIFTIN die weitere
//        Verarbeitung der Daten gesteuert.
//   4. DACOUT (Steuerung des externen D/A-Wandlers)
//        Von diesem Modul werden die Daten- und Kontrollsignale
//        fuer die Ansteuerung des D/A-Wandlers bereitgestellt.
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
//
// Der Entwurf basiert auf einer synchronen, ausschliesslich auf
//  positive Taktflanken reagierenden, Register-Transfer-Logik.
//
//-----------------------------------------------------------------------------

module DAR (DADAT, DACLK, DACSN, DALDN,
            VALID, COPYR, COPYG, IECIN, CLKIN);

  //
  // Port-Deklarationen fuer “Pins”
  //
  output        DADAT;  // Daten fuer seriellen D/A-Wandler
  output        DACLK;  // Datentakt fuer seriellen D/A-Wandler
  output        DACSN;  // Selektionssignal fuer seriellen D/A-Wandler
  output        DALDN;  // Ladesignal fuer seriellen D/A-Wandler
  output        VALID;  // Anzeige eines gueltigen IEC-Signals
  output        COPYR;  // Anzeige des Kopierschutzstatus
  output        COPYG;  // Anzeige der Kopiergeneration
  input         IECIN;  // Eingang fuer das Digital-Audio-Signal
  input         CLKIN;  // Eingang fuer den Arbeitstakt

  //
  // Wire-Deklarationen fuer “Pins”
  //
  wire          DADAT;  // Daten fuer seriellen D/A-Wandler
  wire          DACLK;  // Datentakt fuer seriellen D/A-Wandler
  wire          DACSN;  // Selektionssignal fuer seriellen D/A-Wandler
  wire          DALDN;  // Ladesignal fuer seriellen D/A-Wandler
  wire          VALID;  // Anzeige eines gueltigen IEC-Signals
  wire          COPYR;  // Anzeige des Kopierschutzstatus
  wire          COPYG;  // Anzeige der Kopiergeneration
  wire          IECIN;  // Eingang fuer das Digital-Audio-Signal
  wire          CLKIN;  // Eingang fuer den Arbeitstakt

  //
  // Wire-Deklarationen fuer interne Verbindungen
  //
  wire          DECDA;  // Dekodierte Datenbits von BIPDEC
  wire          SYNCB;  // Bitsynchronisation der Datenbits von BIPDEC
  wire          SYNCD;  // Datenblocksynchronisation von BIPDEC
  wire          PREAM;  // Anzeige einer ‘M’-Praeambel von BIPDEC
  wire          PREAW;  // Anzeige einer ‘W’-Praeambel von BIPDEC
  wire          PREAB;  // Anzeige einer ‘B’-Praeambel von BIPDEC
  wire   [11:0] AUDIO;  // 12-Bit Audio-Daten des Datenblocks
  wire          I_BIT;  // Interpolationsanzeige des Datenblocks
  wire          S_BIT;  // Statusbit des Datenblocks
  wire          ERROR;  // Fehleranzeige fuer Datenblock
  wire          DCOMP;  // Empfangsstatus fuer Datenblock
  wire          LSAMP;  // Anzeige von L-Kanaldaten fuer DACOUT
  wire          RSAMP;  // Anzeige von R-Kanaldaten fuer DACOUT
  wire          LMUTE;  // L-Kanal stummschalten (L-Kanaldaten ungueltig)
  wire          RMUTE;  // R-Kanal stummschalten (R-Kanaldaten ungueltig)

  //
  // Instanzierung der Schaltungs-Module
  //
  BIPDEC  bipdec  (DECDA, SYNCB, SYNCD,
                   PREAM, PREAW, PREAB,
                   IECIN, CLKIN);
  SHIFTIN shiftin (AUDIO, I_BIT, S_BIT, ERROR,
                   DCOMP, DECDA, SYNCD, SYNCB, CLKIN);
  CONTROL control (VALID, COPYR, COPYG, LSAMP, RSAMP, LMUTE, RMUTE,
                   PREAM, PREAW, PREAB, I_BIT, S_BIT, ERROR, DCOMP,
                   CLKIN);
  DACOUT  dacout  (DADAT, DACLK, DACSN, DALDN, AUDIO,
                   LSAMP, RSAMP, LMUTE, RMUTE, CLKIN);

endmodule


//-----------------------------------------------------------------------------
//
// BIPDEC: Biphasen-Decoder mit Praeambel-Detektion
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
//
// Dieses Modul dekodiert ein biphasen-kodiertes Signal und detektiert die
//  (nicht dem Biphasen-Code entsprechenden) Praeambeln der Datenbloecke des
//  Digital-Audio-Formates. Da die Phasenlage des Taktes bezueglich der
//  Daten keinen Einschraenkungen unterliegt, wird das Eingangssignal mit
//  der etwa dreifachen Bitrate der biphasen-Codebits abgetastet und die
//  Synchronisation anhand der Flankenwechsel des abgetasteten Signals
//  vorgenommen.
//
// Die Praeambel-Detektion und die Biphasen-Dekodierung wird unter
//  Verwendung eines 28-Bit Schieberegisters vorgenommen. Die Praeambel-
//  Detektion erfolgt durch einen Flanken-Mustervergleich im Bereich der
//  ersten (aeltesten) 24 Bits. Die Biphasen-Dekodierung erfolgt durch
//  eine Flankendetektion im Bereich der letzten (juengsten) 6 Bits.
// Bei Erkennung einer Praeambel werden neben dem entsprechenden
//  Anzeige-Signal auch die Synchronisations-Signale fuer Datenbits und
//  Audio-Subframes aktiviert. Das Synchronisations-Signal fuer die
//  Datenbits wird ausserdem ueber einen Modulo-6-Zaehler periodisch
//  aktiviert, welcher durch die Praeambel-Detektion synchronisiert wird
//  und durch die Flanken zwischen den Datenbits resynchronisiert wird.
//
// Die Praeambel-Detektion reagiert auf folgende Muster:
//
//   111111??0??0??0??1??001  + Praeambel 'M' (linker Kanal, Frame 1)
//   000000??1??1??1??0??110  - Praeambel 'M' (linker Kanal, Frame 1)
//   ====------======------=
//   111111??0??0??1??0??001  + Praeambel 'W' (rechter Kanal)
//   000000??1??1??0??1??110  - Praeambel 'W' (rechter Kanal)
//   ====------======------=
//   111111??0??1??0??0??001  + Praeambel 'B' (linker Kanal, Frame 2-192)
//   000000??1??0??1??1??110  - Praeambel 'B' (linker Kanal, Frame 2-192)
//   |                    ||
//   |                    |erste biphasen-Codebit-Abtastung
//   aeltestes Bit        juengstes Praeambel-Bit
//
//   Die don't cares ('?') blenden hierbei Bereiche mit unsicherer
//    Signalstabilitaet aus (Phasen-Jitter, Signalflanken-Verschliff)
//
// Die Biphasen-Dekodierung arbeitet wie folgt:
//
//   ?0??0?  =>  0-Bit
//   ?1??1?  =>  0-Bit
//   ?1??0?  =>  1-Bit
//   ?0??1?  =>  1-Bit
//   |    |
//   |     - juengstes Bit
//    - aeltestes Bit im Dekodier-Bereich
//
// Signale und Kodierungen:
//
//   Eingaenge:
//     IECIN:  Digital-Audio-Signal
//     CLKIN:  Arbeitstakt (ungefaehr dreifache Rate der Codebitrate)
//
//   Ausgaenge:
//     DECDA:  dekodierte Datenbits
//     SYNCB:  Synchronisation fuer die Eintaktung eines Datenbits
//              0 = kein gueltiges Datenbit an DECDA
//              1 = Datenbit an DECDA ist gueltig
//     SYNCD:  Synchronisation fuer das 1. Datenblock-Datenbit
//              0 = Datenbit an DECDA ist nicht 1. Datenblock-Datenbit
//              1 = Datenbit an DECDA ist 1. Datenblock-Datenbit
//     PREAM:  Anzeige einer 'M'-Praeambel eines Datenblocks
//              0 = keine 'M'-Praeambel im Schieberegister
//              1 = 'M'-Praeambel im Schieberegister gefunden
//     PREAW:  Anzeige einer 'W'-Praeambel eines Datenblocks
//              0 = keine 'W'-Praeambel im Schieberegister
//              1 = 'W'-Praeambel im Schieberegister gefunden
//     PREAB:  Anzeige einer 'B'-Praeambel eines Datenblocks
//              0 = keine 'B'-Praeambel im Schieberegister
//              1 = 'B'-Praeambel im Schieberegister gefunden
//
//-------------------------------------------------------------------------
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
module BIPDEC (DECDA, SYNCB, SYNCD, PREAM, PREAW, PREAB, IECIN, CLKIN);

  //
  // Port-Deklarationen
  //
  output DECDA;  // Ausgang fuer dekodierte Datenbits
  output SYNCB;  // Ausgang fuer Datenbit-Synchronisation
  output SYNCD;  // Ausgang fuer Datenblock-Synchronisation
  output PREAM;  // Ausgang fuer Anzeige einer 'M'-Praeambel
  output PREAW;  // Ausgang fuer Anzeige einer 'W'-Praeambel
  output PREAB;  // Ausgang fuer Anzeige einer 'B'-Praeambel
  input  IECIN;  // Eingang fuer das Digital-Audio-Signal
  input  CLKIN;  // Eingang fuer den Arbeitstakt

  //
  // Register-Deklarationen fuer Modulschnittstelle
  //
  reg        DECDA;  // Dekodierte Datenbits
  reg        SYNCB;  // Datenbit-Synchronisation
  reg        SYNCD;  // Datenblock-Synchronisation
  reg        PREAM;  // Anzeige einer 'M'-Praeambel
  reg        PREAW;  // Anzeige einer 'W'-Praeambel
  reg        PREAB;  // Anzeige einer 'B'-Praeambel

  //
  // Wire-Deklarationen fuer Modulschnittstelle
  //
  wire   IECIN;  // Digital-Audio-Signal
  wire   CLKIN;  // Arbeitstakt

  //
  // Deklaration der Arbeitsregister
  //
  reg [27:0] DECSR;  // 28-Bit Schieberegister fuer Dekodierung
  reg [ 2:0] MOD6C;  // Zaehler-Register fuer Modulo-6 Zaehler
  reg [ 2:0] PREAC;  // Praeambel-Detektionscode


  //
  // Anfangs-Initialisierung der Register, nicht synthetisierbar
  // synopsys translate_off
  //
  initial begin
    DECDA = 1'b0;
    SYNCB = 1'b0;
    SYNCD = 1'b0;
    PREAM = 1'b0;
    PREAW = 1'b0;
    PREAB = 1'b0;
    DECSR = 28'b0;
    MOD6C = 3'b0;
    PREAC = 3'b0;
  end // synopsys translate_on

  //
  // Ausgang fuer dekodierte Datenbits aktualisieren
  //
  always @(posedge CLKIN) begin
    DECDA <= DECSR[4] ^ DECSR[1];          // Dekodierung
  end

  //
  // Datenbit-Synchronisation melden
  //
  always @(posedge CLKIN) begin
    SYNCB <= (|PREAC) | ~(|MOD6C) | (~(|(MOD6C ^ 3'b101)) & ^DECSR[6:5]);
  end

  //
  // Subframe-Synchronisation melden
  //
  always @(posedge CLKIN) begin
    SYNCD <= |PREAC;                       // Synchronisition melden
  end
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
  //
  // Praeambel-Anzeigen aktualisieren
  //
  always @(posedge CLKIN) begin
    {PREAM, PREAW, PREAB} <= PREAC;        // Praeambeln anzeigen
  end

  //
  // Dekodier-Schieberegister aktualisieren
  //
  always @(posedge CLKIN) begin
    DECSR[27:0] <= {DECSR[26:0], IECIN};   // DECSR weiterschieben
  end

  //
  // Modulo-6 Zaehler fuer Datenbit-Synchronisation aktualisieren
  //
  always @(posedge CLKIN) begin
    casez ({|PREAC, MOD6C, ^DECSR[6:5]})         // Mod-6-Zaehler aktualisieren
      5'b1????: MOD6C <= 3'b001;
      5'b0000?: MOD6C <= 3'b001;
      5'b00010: MOD6C <= 3'b010;
      5'b00011: MOD6C <= 3'b001;
      5'b0010?: MOD6C <= 3'b011;
      5'b0011?: MOD6C <= 3'b100;
      5'b0100?: MOD6C <= 3'b101;
      5'b01010: MOD6C <= 3'b000;
      5'b01011: MOD6C <= 3'b001;
      default:  MOD6C <= 3'b000;
    endcase
  end

  //
  // Praeambel-Detektion im Dekodier-Schieberegister vornehmen
  //
  always @(DECSR) begin
    casez (DECSR[27:5])
      23'b1111_11??0?_?0??0?_?1??00_1: PREAC = 3'b100;
      23'b0000_00??1?_?1??1?_?0??11_0: PREAC = 3'b100;
      23'b1111_11??0?_?0??1?_?0??00_1: PREAC = 3'b010;
      23'b0000_00??1?_?1??0?_?1??11_0: PREAC = 3'b010;
      23'b1111_11??0?_?1??0?_?0??00_1: PREAC = 3'b001;
      23'b0000_00??1?_?0??1?_?1??11_0: PREAC = 3'b001;
      default:                         PREAC = 3'b000;
    endcase
  end

endmodule


//-----------------------------------------------------------------------------
//
// SHIFTIN: Empfangs-Schieberegister fuer Datenbloecke
//
// In diesem Modul wird der serielle Datenstrom eines Digital-Audio-Signals
//  in die Datenbloecke der einzelnen Audio-Abtastungen umgewandelt.
// Hierbei wird auch die Gueltigkeit der Datenblock-Paritaet ueberprueft.
//
// Die Umwandlung erfolgt unter Verwendung eines 29-Bit Schieberegisters,
//  welches neben einem Markierungsbit fuer die Anzeige des vollstaendigen
//  Empfangs die der Praeambel (4-Bit) folgenden Datenbits (28-Bits) des
//  Datenblocks (32-Bit) aufnimmt.
// Die Daten werden in das Register uebernommen, wenn bei einer positiven
//  Taktflanke des 6-fachen Datenbit-Taktes die Datenlock- oder Bit-
//  Synchronisationsanzeige aktiv (=1) ist.
// Die Eintaktung erfolgt beim MSB des Registers, so dass die Audio-Daten
//  nach vollstaendiger Eintaktung des Datenblocks im Bereich [23:4] liegen
//  (mit Audio-MSB bei [23], da das Audio-LSB zuerst uebertragen wird).
// Wenn der Datenblock vollstaendig eingetaktet ist (DCOMP = 1),
//  liegen seine Daten uebernahmebereit an den Modul-Ausgaengen an.
// Die Audio-Abtastungsdaten liegen hierbei, wie auch im Datenblock,
//  in Zweierkomplementdarstellung vor und geben die PCM-Amplitude
//  relativ zur 0-Amplitude an.
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
//
// Signale und Kodierungen:
//
//   Eingaenge:
//     DECDA:  dekodiertes Digital-Audio Signal
//     SYNCB:  Synchronisation fuer die Eintaktung eines Datenbits
//              0 = kein gueltiges Datenbit an DECDA
//              1 = Datenbit an DECDA ist gueltig
//     SYNCD:  Synchronisation fuer die Eintaktung des 1. Datenblock-Datenbits
//              0 = Datenbit an DECDA ist nicht 1. Datenblock-Datenbit
//              1 = Datenbit an DECDA ist 1. Datenblock-Datenbit
//     CLKIN:  Arbeitstakt
//
//   Ausgaenge:
//     AUDIO:  12-Bit Audio-Daten in Zweierkomplementdarstellung
//     I_BIT:  Interpolationsbit (1 = interpolierte Audio-Daten, 0 = sonst)
//     S_BIT:  Statusbit (192 Bloecke -> 'Channel status block')
//     ERROR:  Ergebnis der EVEN-Parity Pruefung (1 = Fehler, 0 = sonst)
//     DCOMP:  Status des Datenblock-Empfangs (1 = vollstaendig, 0 = sonst)
//
//------------------------------------------------------------------------------

module SHIFTIN (AUDIO, I_BIT, S_BIT, ERROR,
                DCOMP, DECDA, SYNCD, SYNCB, CLKIN);

  //
  // Port-Deklarationen
  //
  output [11:0] AUDIO;  // Ausgang fuer 12-Bit Audio-Daten des Datenblocks
  output        I_BIT;  // Ausgang fuer Interpolationsbit des Datenblocks
  output        S_BIT;  // Ausgang fuer Statusbit des Datenblocks
  output        ERROR;  // Ausgang fuer Fehleranzeige (Paritaetspruefung)
  output        DCOMP;  // Ausgang fuer Datenblock-Empfangsstatus
  input         DECDA;  // Eingang fuer dekodierte Datenbits
  input         SYNCD;  // Eingang fuer Block-Synchronisation
  input         SYNCB;  // Eingang fuer Bit-Synchronisation
  input         CLKIN;  // Eingang fuer Arbeitstakt

  //
  // Wire-Deklarationen fuer Modulschnittstelle
  //
  wire   [11:0] AUDIO;  // 12-Bit Audio-Daten des Datenblocks
  wire          I_BIT;  // Interpolationsbit des Datenblocks
  wire          S_BIT;  // Statusbit des Datenblocks
  wire          DCOMP;  // Datenblock-Empfangsstatus
  wire          DECDA;  // dekodierte Datenbits
  wire          SYNCD;  // Datenblock-Synchronisation
  wire          SYNCB;  // Bit-Synchronisation
  wire          CLKIN;  // Arbeitstakt

  //
  // Deklaration des Arbeitsregisters
  //
  reg    [28:0] BLOCK;  // Datenblock-Schieberegister
  reg           ERROR;  // Paritats-Pruefungsregister

  //
  // Continuous-Assignments fuer Modulschnittstelle
  //
  assign AUDIO = BLOCK[24:13]; // 12-Bit Audio-Daten
  assign I_BIT = BLOCK[25];    // Interpolationsbit
  assign S_BIT = BLOCK[27];    // Statusbit
  assign DCOMP = BLOCK[0];     // Empfangsstatus (1=vollstaendig)


  //
  // Anfangs-Initialisierung der Register, nicht synthetisierbar
  // synopsys translate_off
  //
  initial begin
    BLOCK = 29'b0;
    ERROR = 1'b0;
  end // synopsys translate_on
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
  //
  // Datenblock-Schieberegister aktualisieren
  //
  always @(posedge CLKIN) begin
    if (SYNCD == 1'b1)
      BLOCK <= {DECDA, 1'b1, 27'b0};             // neuer Datenblock
    else
      if ((SYNCB == 1'b1) && (BLOCK[0] == 1'b0))
        BLOCK[28:0] <= {DECDA, BLOCK[28:1]};     // Datenblock eintakten
  end

  //
  // Paritaets-Pruefungsregister aktualisieren
  //
  always @(posedge CLKIN) begin
    if (SYNCD == 1'b1)
      ERROR <= DECDA;                        // neuen Datenblock pruefen
    else
      if ((SYNCB == 1'b1) && (BLOCK[0] == 1'b0))
        ERROR <= ERROR ^ DECDA;              // "Pruefsumme" bilden
  end

endmodule


//-----------------------------------------------------------------------------
//
// CONTROL: Kontroll-Logik fuer die Verarbeitung von Audio-Datenbloecken
//
// In diesem Modul werden die Statusinformationen der Datenbloecke
//  extrahiert und die weitere Verarbeitung der Daten gesteuert.
//
// Die Steuerung basiert auf einem 5-Bit Zaehler, in welchem die
//  Blockpaare abgezaehlt werden, und einigen Statusregistern, in denen
//  die Praeambel-Codes und die auszugebenden Statusdaten gesichert werden.
// Der Zaehler ist ein Countdown-Zaehler ohne automatischen Neustart.
// Er ist ausreichend fuer diese Anwendung, da alle relevanten Statusbits
//  in den Datenblock-Paaren 0 bis 15 auftreten.
//
// Bei Erkennung eines Praeambel-Codes wird das entsprechende Statusregister
//  gesetzt. Handelt es sich um eine 'B'-Praeambel (1. Block einer Sequenz),
//  so wird ausserdem der 5-Bit Zaehler auf den Wert 16 gesetzt.
// Der Zaehler wird bei Erkennung einer 'M'-Praeaembel (Datenblock "links")
//  dekrementiert, sofern er einen Wert ungleich 0 besitzt.
//
// Folgende Audio-Statusbits werden in Register uebernommen:
//  Bit  0:  Audio-Uebertragungsmodus
//            0 = SPDIF (Consumer)
//            1 = AES3  (Professional)
//  Bit  1:  Kopierschutz-Status
//            0 = Kopierschutz aktiv
//            1 = kein Kopierschutz
//  Bit 15:  Kopiergeneration
//            0 = Original
//            1 = Kopie
//
// Je nach Inhalt der Praeambel-Statusregister wird bei einem vollstaendig
//  empfangenen Datenblock das entsprechende Kontrollsignal fuer die
//  Weiterverarbeitung des Audio-Daten in den Ausgabekanaelen aktiviert
//  ('M'/'B': linker Kanal = LSAMP, 'W': rechter Kanal = RSAMP).
// Kann das Signal nicht in den Ausgabekanaelen verarbeitet werden
//  (AES3-Signal, Empfangsfehler oder ungueltiger Abtastwert),
//  so wird fuer den entsprechenden Ausgabekanal zusaetzlich eine
//  Stummschaltung signalisiert (ueber LMUTE bzw. RMUTE).
//
// Signale und Kodierungen:
//
//   Eingaenge:
//     PREAM:  Anzeige einer 'M'-Praeambel eines Datenblocks
//              0 = keine 'M'-Praeambel
//              1 = 'M'-Praeambel
//     PREAW:  Anzeige einer 'W'-Praeambel eines Datenblocks
//              0 = keine 'W'-Praeambel
//              1 = 'W'-Praeambel
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
//     PREAB:  Anzeige einer 'B'-Praeambel eines Datenblocks
//              0 = keine 'B'-Praeambel
//              1 = 'B'-Praeambel
//     I_BIT:  Interpolationsbit des aktuellen Datenblocks
//              0 = Datenblock enthaelt fehlerfreie Audio-Daten
//              1 = Datenblock enthaelt interpolierte Audio-Daten
//     S_BIT:  Statusbit des aktuellen Datenblocks
//     ERROR:  Fehlerstatus (Paritaetspruefung) des aktuellen Datenblocks
//              0 = kein Empfangsfehler
//              1 = Empfangsfehler
//     DCOMP:  Datenblock-Empfangsstatus
//              0 = Datenblock noch nicht vollstaendig empfangen
//              1 = Datenblock wurde vollstaendig empfangen
//     CLKIN:  Arbeitstakt
//
//   Ausgaenge:
//     VALID:  Anzeige fuer gueltiges SPDIF Digital-Audio-Signal
//              0 = es liegt kein gueltiges SPDIF Signal an
//              1 = ein gueltiges SPDIF Signal wird empfangen
//     COPYR:  Anzeige des Kopierschutzstatus des SPDIF Signals
//              0 = Audio-Daten sind nicht kopiergeschuetzt
//              1 = Audio-Daten unterliegen einem Kopierschutz
//     COPYG:  Anzeige der Kopiergeneration des SPDIF Signals
//              0 = Audio-Daten liegen im Original vor
//              1 = Audio-Daten liegen als Kopie vor
//     LSAMP:  Anzeige von empfangenen Daten fuer den linken Stereokanal
//              0 = empfangene Daten sind nicht fuer den linken Kanal
//              1 = empfangene Daten sind fuer den linken Kanal
//     RSAMP:  Anzeige von empfangenen Daten fuer den rechten Stereokanal
//              0 = empfangene Daten sind nicht fuer den rechten Kanal
//              1 = empfangene Daten sind fuer den rechten Kanal
//     LMUTE:  Anzeige von gueltigen Audio-Daten fuer den linken Stereokanal
//              0 = Ausgabe ueber linken Stereokanal aktivieren
//              1 = linken Stereokanal stummschalten (ungueltige Daten)
//     RMUTE:  Anzeige von gueltigen Audio-Daten fuer den rechten Stereokanal
//              0 = Ausgabe ueber rechten Stereokanal aktivieren
//              1 = rechten Stereokanal stummschalten (ungueltige Daten)
//
//-----------------------------------------------------------------------------

module CONTROL (VALID, COPYR, COPYG, LSAMP, RSAMP, LMUTE, RMUTE,
                PREAM, PREAW, PREAB, I_BIT, S_BIT, ERROR, DCOMP, CLKIN);

  //
  // Port-Deklarationen
  //
  output        VALID;  // Ausgang fuer Anzeige eines gueltigen SPDIF-Signals
  output        COPYR;  // Ausgang fuer Anzeige des Kopierschutz-Status
  output        COPYG;  // Ausgang fuer Anzeige der Kopiergeneration
  output        LSAMP;  // Ausgang fuer Anzeige eines "linken" Abtastwertes
  output        RSAMP;  // Ausgang fuer Anzeige einer "rechten" Abtastwertes
  output        LMUTE;  // Ausgang fuer Anzeige ungueltiger "linker" Daten
  output        RMUTE;  // Ausgang fuer Anzeige ungueltiger "rechter" Daten
  input         PREAM;  // Eingang fuer Anzeige einer 'M'-Praeambel
  input         PREAW;  // Eingang fuer Anzeige einer 'W'-Praeambel
  input         PREAB;  // Eingang fuer Anzeige einer 'B'-Praeambel
  input         I_BIT;  // Eingang fuer Interpolationsbit des Datenblocks
  input         S_BIT;  // Eingang fuer Statusbit des Datenblocks
  input         ERROR;  // Eingang fuer Fehleranzeige fuer Datenblock
  input         DCOMP;  // Eingang fuer Empfangsstatus fuer Datenblock
  input         CLKIN;  // Eingang fuer Arbeitstakt

  //
  // Wire-Deklarationen fuer Modulschnittstelle
  //
  wire          PREAM;  // Anzeige einer 'M'-Praeambel
  wire          PREAW;  // Anzeige einer 'W'-Praeambel
  wire          PREAB;  // Anzeige einer 'B'-Praeambel
  wire          I_BIT;  // Interpolationsbit des Datenblocks
  wire          S_BIT;  // Statusbit des Datenblocks
  wire          ERROR;  // Fehleranzeige fuer Datenblock
  wire          DCOMP;  // Empfangsstatus fuer Datenblock
  wire          CLKIN;  // Arbeitstakt
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
//
// Register-Deklarationen fuer Modulschnittstelle
//
reg           VALID;  // Anzeige eines gueltigen SPDIF-Signals
reg           COPYR;  // Anzeige des Kopierschutz-Status
reg           COPYG;  // Anzeige der Kopiergeneration
reg           LSAMP;  // Anzeige eines “linken” Abtastwertes
reg           RSAMP;  // Anzeige eines “rechten” Abtastwertes
reg           LMUTE;  // Anzeige ungueltiger “linker” Daten
reg           RMUTE;  // Anzeige ungueltiger “rechter” Daten

//
// Deklaration der Arbeitsregister
//
reg           PRERM;  // Register zur Sicherung der ‘M’-Praeambel-Anzeige
reg           PRERW;  // Register zur Sicherung der ‘W’-Praeambel-Anzeige
reg           PRERB;  // Register zur Sicherung der ‘B’-Praeambel-Anzeige
reg           TMODE;  // Register zur Sicherung des Uebertragungsmodus
reg           ADVAL;  // Register zur Sicherung der Audio-Daten-Gueltigkeit
reg    [4:0]  BPCNT;  // Zaehler-Register fuer Blockpaare


//
// Anfangs-Initialisierung der Register, nicht synthetisierbar
// synopsys translate_off
//
initial begin
  VALID = 1’b0;
  COPYR = 1’b0;
  COPYG = 1’b0;
  LSAMP = 1’b0;
  RSAMP = 1’b0;
  LMUTE = 1’b0;
  RMUTE = 1’b0;
  PRERM = 1’b0;
  PRERW = 1’b0;
  PRERB = 1’b0;
  TMODE = 1’b0;
  ADVAL = 1’b0;
  BPCNT = 5’b0;
end // synopsys translate_on

//
// Register zur Sicherung der Datenblock-Praeambel-Anzeigen aktualisieren
//
always @(posedge CLKIN) begin
  if (|{PREAM, PREAW, PREAB} == 1’b1)
    {PRERM, PRERW, PRERB} <= {PREAM, PREAW, PREAB};  // neue Anzeige
end

//
// Zaehler-Register fuer Datenblockpaare aktualisieren
//
always @(posedge CLKIN) begin
  if (PREAB)
    BPCNT <= 16;                        // neue Blockpaar-Sequenz
  else
    if (PREAM & (|BPCNT))
      BPCNT <= BPCNT - 1;               // neues Blockpaar
end

//
// ‘Valid Consumer Audio’-Anzeige (VALID) aktualisieren
//
always @(TMODE or ADVAL) begin
  VALID = TMODE & ADVAL;
end

//
// Statusbit fuer Uebertragungsmodus aktualisieren
//
always @(posedge CLKIN) begin
  if ((BPCNT == 16) && ({DCOMP, ERROR} == 2’b10))
    TMODE <= ~S_BIT;                        // Statusbit aus 1. Paar
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
  end

  //
  // Statusbit fuer Gueltigkeit der Audio-Daten aktualisieren
  //
  always @(posedge CLKIN) begin
    if ((BPCNT == 15) && ({DCOMP, ERROR} == 2'b10))
      ADVAL <= ~S_BIT;                        // Statusbit aus 2. Paar
  end

  //
  // 'Copyright'-Anzeige (COPYR) aktualisieren
  //
  always @(posedge CLKIN) begin
    if ((BPCNT == 14) && ({DCOMP, VALID, ERROR} == 3'b110))
      COPYR <= ~S_BIT;                     // Statusbit aus 3. Paar
  end

  //
  // 'Copy generation'-Anzeige (COPYG) aktualisieren
  //
  always @(posedge CLKIN) begin
    if ((BPCNT == 1) && ({DCOMP, VALID, ERROR} == 3'b110))
      COPYG <= S_BIT & COPYR;            // Statusbit aus 15. Paar
  end

  //
  // Anzeige fuer "linke" Abtastwerte (left samples) aktualisieren
  //
  always @(posedge CLKIN) begin
    LSAMP <= (PRERM | PRERB) & DCOMP & ~(|{PREAM, PREAW, PREAB});
  end

  //
  // Anzeige fuer "rechte" Abtastwerte (right samples) aktualisieren
  //
  always @(posedge CLKIN) begin
    RSAMP <= PRERW & DCOMP & ~(|{PREAM, PREAW, PREAB});
  end

  //
  // Stummschaltung fuer ungueltige "linke" Abtastwerte aktualisieren
  //
  always @(posedge CLKIN) begin
    case ({(PRERM | PRERB), DCOMP, ERROR, I_BIT, VALID})
      5'b11001: LMUTE <= 1'b0;            // Abtastwert gueltig
      default:  LMUTE <= 1'b1;            // Abtastwert ungueltig
    endcase
  end

  //
  // Stummschaltung fuer ungueltige "rechte" Abtastwerte  aktualisieren
  //
  always @(posedge CLKIN) begin
    case ({PRERW, DCOMP, ERROR, I_BIT, VALID})
      5'b11001: RMUTE <= 1'b0;            // Abtastwert gueltig
      default:  RMUTE <= 1'b1;            // Abtastwert ungueltig
    endcase
  end

endmodule


//-----------------------------------------------------------------------------
//
// DACOUT: Steuerung des externen D/A-Wandlers
//
// In diesem Modul wird die Ausgabe der Audio-Abtastwerte
//  an den externen Digital-Analog-Converter (DAC) gesteuert.
//
// Das Modul besteht aus einem Schieberegister fuer die serielle
//  Datenuebergabe an den DAC, einem Zustandszaehlerregister fuer
//  die Steuerung der Datenuebergabe und einem Takt-Teiler fuer die
//  Erzeugung des Datentaktes des DACs.
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
//
// Liegt ein gueltiger Abtastwert am Modul an, so wird dieser in das
//  Ausgangsschieberegister uebernommen, wobei die fuer den DAC
//  notwendigen Steuercodes hinzugefuegt werden. Das hoechstwertigste
//  Bit des Wertes wird hierbei invertiert, um eine Umwandlung von
//  der Zweierkomplementdarstellung in die Offset-Darstellung
//  durchzufuehren (entspricht Addition von $800). Des weiteren
//  wird der Zustandszaehler fuer die Datenuebergabe auf seinen
//  Startwert gesetzt (17).
// Sobald die Uebernahme des Abtastwertes in das Ausgangsschieberegister
//  abgeschlossen ist, wird der Zaehler mit CLKIN/4 dekrementiert und
//  synchron dazu das Schieberegister ausgetaktet, bis der Zaehler den
//  Endwert erreicht (0). In den Zustaenden 16-1 wird der DAC selektiert,
//  im ersten 0-Zustands-Takt nach dem 1-Zustand aktualisiert.
//
// Signale und Kodierungen:
//
//   Eingaenge:
//     AUDIO:  12-Bit Audio-Daten aus einem Datenblock (Zweierkomplement)
//     LSAMP:  Anzeige von Daten fuer den linken Stereokanal
//              0 = Daten sind nicht fuer den linken Kanal
//              1 = Daten sind fuer den linken Kanal
//     RSAMP:  Anzeige von Daten fuer den rechten Stereokanal
//              0 = Daten sind nicht fuer den rechten Kanal
//              1 = Daten sind fuer den rechten Kanal
//     LMUTE:  Anzeige von gueltigen Daten fuer den linken Stereokanal
//              0 = Ausgabe ueber linken Stereokanal aktivieren
//              1 = linken Stereokanal stummschalten (ungueltige Daten)
//     RMUTE:  Anzeige von gueltigen Daten fuer den rechten Stereokanal
//              0 = Ausgabe ueber rechten Stereokanal aktivieren
//              1 = rechten Stereokanal stummschalten (ungueltige Daten)
//     CLKIN:  Arbeitstakt
//
//   Ausgaenge:
//     DADAT:  Daten fuer seriellen D/A-Wandler
//     DACLK:  Datentakt fuer seriellen D/A-Wandler (posedge-Datenuebergabe)
//     DACSN:  Selektion fuer seriellen D/A-Wandler
//              0 = DAC ist selektiert
//              1 = DAC ist deselektiert
//     DALDN:  Load-Signal fuer seriellen D/A-Wandler
//              1->0  = Daten im Eingangsschiebregister des DAC umsetzen
//              sonst = Analog-Daten am DAC-Ausgang halten
//
//-----------------------------------------------------------------------------

module DACOUT (DADAT, DACLK, DACSN, DALDN, AUDIO, LSAMP, RSAMP,
               LMUTE, RMUTE, CLKIN);

  //
  // Port-Deklarationen
  //
  output        DADAT;  // Ausgang fuer Daten fuer seriellen D/A-Wandler
  output        DACLK;  // Ausgang fuer Datentakt fuer seriellen D/A-Wandler
  output        DACSN;  // Ausgang fuer Selektion fuer seriellen D/A-Wandler
  output        DALDN;  // Ausgang fuer Load-Signal des seriellen D/A-Wandlers
  input  [11:0] AUDIO;  // Eingang fuer 12-Bit Audio-Daten aus einem Datenblock
  input         LSAMP;  // Eingang fuer Anzeige eines “linken” Abtastwertes
  input         RSAMP;  // Eingang fuer Anzeige eines “rechten” Abtastwertes
  input         LMUTE;  // Eingang fuer Stummschaltung des linken Kanals
  input         RMUTE;  // Eingang fuer Stummschaltung des rechten Kanals
  input         CLKIN;  // Eingang fuer Arbeitstakt

  //
  // Wire-Deklarationen fuer Modulschnittstelle
  //
  wire          DACSN;  // Selektions-Signal fuer seriellen D/A-Wandler
  wire          DALDN;  // Load-Signal fuer seriellen D/A-Wandler
  wire   [11:0] AUDIO;  // 12-Bit Audio-Daten aus einem Datenblock
  wire          LSAMP;  // Anzeige eines “linken” Abtastwertes
  wire          RSAMP;  // Anzeige eines “rechten” Abtastwertes
  wire          LMUTE;  // linken Kanal stummschalten (Abtastwert ungueltig)
  wire          RMUTE;  // rechten stummschalten (Abtastwert ungueltig)
  wire          CLKIN;  // Arbeitstakt
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
//
// Register-Deklarationen fuer Modulschnittstelle
//
reg           DADAT;  // Daten fuer seriellen D/A-Wandler
reg           DACLK;  // Datentakt fuer seriellen D/A-Wandler
reg           DACSP;  // Inv. Selektions-Signal fuer seriellen D/A-Wandler
reg           DALDP;  // Inv. Load-Signal fuer seriellen D/A-Wandler

//
// Deklaration der Arbeitsregister
//
reg    [15:0] SHIFT;  // Schieberegister fuer serielle DAC-Ausgangsdaten
reg    [ 4:0] STATE;  // Zustandsregister fuer DAC-Ansteuerung
reg    [ 1:0] CLKD4;  // Taktteilerregister fuer CLKIN/4

//
// Continuous-Assignments fuer Modulschnittstelle
//  (notwendig fuer XC3000-Synthese, da hierbei 0-RESET der Register)
//
assign DACSN = ~DACSP;     // Selektions-Signal fuer seriellen D/A-Wandler
assign DALDN = ~DALDP;     // Load-Signal fuer seriellen D/A-Wandler

//
// Umwandlung des Audio-Samples von der Zweierkomplementdarstellung
//  in die Offset-Darstellung (positive Amplitude von 0-Basis aus)
//
wire   [11:0] AMPLI = {~AUDIO[11], AUDIO[10:0]};


//
// Anfangs-Initialisierung der Register, nicht synthetisierbar
// synopsys translate_off
//
initial begin
  DADAT = 1'b0;
  DACLK = 1'b0;
  DACSP = 1'b0;
  DALDP = 1'b0;
  SHIFT = 16'b0;
  STATE = 5'b0;
  CLKD4 = 2'b0;
end // synopsys translate_on

//
//  Taktteilerregister aktualisieren
//
always @(posedge CLKIN) begin
  CLKD4 <= CLKD4 + 1;    // Zaehler erhoehen
end

//
// Zustandsregister fuer DAC-Ansteuerung aktualisieren
//
always @(posedge CLKIN) begin
  if (((LSAMP & ~LMUTE) | (RSAMP & ~RMUTE)) & (CLKD4[1:0] == 2'b11)) begin
    STATE <= 5'b10001;  // neues Audio-Sample
  end
  else begin
    if ((LSAMP | RSAMP | ~(&CLKD4) | ~(|STATE)) == 1'b0) begin
      STATE <= STATE - 1;  // Audio-Sample austakten
    end
  end
end


//
// Schieberegister fuer DAC-Daten aktualisieren
//
always @(posedge CLKIN) begin
  if (((LSAMP & ~LMUTE) | (RSAMP & ~RMUTE)) & (&CLKD4)) begin  // neues
    SHIFT <= {1'b1, RSAMP, LSAMP, 1'b0, AMPLI[11:0]};    // Sample
  end
  else begin
    if ((&CLKD4) & (|STATE) & (|(STATE ^ 5'b10001))) begin
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
        SHIFT <= {SHIFT[14:0], 1'b0};   // Audio-Sample austakten
      end
    end
  end

  //
  // Datentakt fuer seriellen D/A-Wandler aktualisieren
  //
  always @(posedge CLKIN) begin
    DACLK <= CLKD4[1];
  end

  //
  // Daten fuer seriellen D/A-Wandler aktualisieren
  //
  always @(posedge CLKIN) begin
    DADAT <= SHIFT[15];
  end

  //
  // Invertiertes Selektions-Signal fuer externen DAC aktualisieren
  //
  always @(posedge CLKIN) begin
    DACSP <= |STATE & |(STATE ^ 5'b10001);
  end

  //
  // Invertiertes Lade-Signal fuer Umsetzer des DAC aktualisieren
  //
  always @(posedge CLKIN) begin
    DALDP <= ~(|STATE) & DACSP;
  end

endmodule
```

Listing B.4: Digital-Audio-Receiver in RTL-Verilog nach [Blinze97]

```
// ------------------------------------------------------------------------
// 'Digital-Audio-Receiver'
// Akustische Wiedergabe eines digitalen Audio-Signals eines Musik-CD-Players
// ueber einen (externen) Digital-Analog-Konverter.
//
// Nach der Aufgabenstellung des VLSI-Entwurfspraktikums im
// Sommersemester 1997
//
// Mixed-Mode-Beschreibung auf Register-Transfer-Ebene als Grundlage fuer
// den Schematics-Entwurf und auch zur spaeteren RTL-Synthese mit
// Synopsys HDL-Compiler for Verilog.
//
// Autor: Arne Friedrichs, 12/97
// ------------------------------------------------------------------------



module dar (CLKIN, IECIN, DADAT, DACLK, DACSN, DALDN, VALID, COPYR, COPYG);
// ------------------------------------------------------------------------
// Top-Level-Modul des DAR, nur Modulinstanzen
//
// CLKIN    Taktsignal (16MHz zwingend)
// IECIN    Digital-Audio-Signal des CD-Players (0V/5V-Uebertragungspegel)
// Schnittstelle zum DAC AD8522:
// DACLK    Taktsignal fuer DAC: 1/2 CLKIN = 8 MHz
// DADAT    Serieller Dateneingang des DAC
// DACSN    Freigabe fuer das Eingangsschieberegister des DAC, active low
// DALDN    Freigabesignal fuer DA-Wandlung im DAC, active low
// Weitere Ausgangssignale:
// VALID    zeigt gueltiges SPDIF-Signal an
// COPYR    zeigt kopiergeschuetzte Daten an
// COPYG    Kopiergeneration der Daten (Original/Kopie)
// ------------------------------------------------------------------------
```

Listing B.5: Digital-Audio-Receiver in RTL-Verilog nach [Friedr98]

```
   input             CLKIN,   // Takteingang
                     IECIN;   // Digital-Audio-Daten des CD-Players
   output            DADAT,   // serielle Daten an DAC
                     DACLK,   // Ausgabetakt fuer DAC
                     DACSN,   // Chip Select des DAC, active low
                     DALDN,   // Load-Signal des DAC, active low
                     VALID,   // 1, wenn gueltiges SPDIF-Signal an IECIN
                     COPYR,   // 1 bei kopiergeschuetzten Daten
                     COPYG;   // 0 = Original, 1 = Kopie

   wire              BIT,     // decodiertes Datenbit des Dig.-Audio-Signals
                     BITRDY,  // BIT-Takt, jeweils 1 Taktzyklus lang gesetzt
                     PREAMB,  // Praeambel erkannt
                     LFTCH,   // Daten sind fuer den linken Kanal
                     RGTCH,   // Daten sind fuer den rechten Kanal
                     SEQBGN,  // Anfang einer Sequenz von Datenbloecken
                     DATARDY, // Datenblock steht bereit
                     PARITY,  // zur Paritaetspruefung
                     STBIT,   // Statusbit
                     INTERPOL,// zeigt interpolierte Daten an
                     STRTOUT; // Ausgabe an DAC starten
   wire     [27:0]   DATABLK; // gelesener Datenblock des Dig.-Audio-Signals
   wire     [11:0]   AMPLITUDE;  // digitaler Wert der Signalamplitude

   // Position der Signale im Datenblock:
   assign STBIT = DATABLK[1];
   assign INTERPOL = DATABLK[3];
   assign AMPLITUDE = DATABLK[15:4];

   // Modul-Instanzen

   bipdec  BIPDEC  (CLKIN, IECIN, BIT, BITRDY, PREAMB, LFTCH, RGTCH, SEQBGN);
   shiftin SHIFTIN (CLKIN, BIT, BITRDY, PREAMB, DATABLK, DATARDY, PARITY);
   dacout  DACOUT  (CLKIN, STRTOUT, LFTCH, RGTCH, AMPLITUDE, DACLK, DADAT,
                    DACSN, DALDN);
   control CONTROL (CLKIN, DATARDY, PARITY, INTERPOL, SEQBGN, LFTCH, STBIT,
                    STRTOUT, VALID, COPYR, COPYG);

endmodule
// ---------------------------------------------------------------------------



module sync (CLK, IECIN, CODEBIT, CBITRDY);
// ---------------------------------------------------------------------------
// Synchronisation des Digital-Audio-Signals
//
// Nach jeweils 3 Takten wird ein Codebit abgetastet, wobei
// jedoch der Abtastzaehler mit jeder Flanke des Digital-Audio-Signals
// neu synchronisiert wird.
// Da das DA-Signal aufgrund der Codierung nach maximal 3 Codebits seinen
// Pegel wechselt, muss zur fehlerfreien Abtastung sichergestellt sein, dass
// die 'Dauer' von 3 Codebits zwischen 8 und 9 Takten liegt.
// Geht man sicherheitshalber von 8,5 Taktzyklen aus, ergibt sich
// bei einer Uebertragungsrate des DA-Signals von ca. 5,64 Mbit/s
// eine zuverlaessige Taktfrequenz von 16 MHz.
// ---------------------------------------------------------------------------

   input             CLK,     // Takteingang
                     IECIN;   // Digital-Audio-Signal
   output            CODEBIT, // Wert des Biphasen-Codebits
                     CBITRDY; // 1 wenn CODEBIT gueltig (1 Taktzyklus aktiv)

   wire              EDGEDETECT; // Flanke im DA-Signal erkannt

   reg      [1:0]    COUNT;   // Abtastzaehler
   reg      [1:0]    SR;      // 2-Bit-Schieberegister zur Flankenerkennung


   // synopsys translate_off
   initial begin              // FPGA-Initialisierung durch GR
      COUNT = 0;
      SR = 0;
   end
```

Listing B.5: Digital-Audio-Receiver in RTL-Verilog nach [Friedr98]

```
   // synopsys translate_on


   // bei einer Flanke im Signal sind die Bits im Schieberegister verschieden
   assign EDGEDETECT = (SR[0] != SR[1]);

   // Codebit nach jeweils 3 Takten oder einer Flanke im Biphasensignal
   assign CBITRDY = (COUNT == 2) || EDGEDETECT;

   // das aktuelle Code-Bit befindet sich dann im oberen Bit des SR
   assign CODEBIT = SR[1];


   always @(posedge CLK) begin

      SR[1:0] <= { SR[0], IECIN };
      // bei jedem Takt neues Bit einschieben

      if (CBITRDY)
         COUNT <= 0;          // Wenn Bit erkannt, Abtastzaehler ruecksetzen,
      else
         COUNT <= COUNT + 1;  // sonst hochzaehlen
   end

endmodule
// ---------------------------------------------------------------------------



module decode (CLK, CODEBIT, CBITRDY, DABIT, DABITRDY, PREAMB, LFTCH, RGTCH,
               SEQBGN);
// ---------------------------------------------------------------------------
// Decodierung des Biphasencode-Signals inklusive Erkennung von Praeambelcodes
//
// Zur Aufnahme einer gesamten Praeambel ist ein 8-Bit-Schieberegister noetig.
// Die Biphasen-Decodierung wird mit den unteren beiden Bits des
// Schieberegisters durchgefuehrt. Durch Halbierung des Codebit-Taktes wird
// der Datentakt gewonnen, der mit der Praeambelerkennung synchronisiert wird.
// ---------------------------------------------------------------------------

   input             CLK,     // Takteingang
                     CODEBIT, // Codebit-Datum
                     CBITRDY; // Codebit-Takt
   output            DABIT,   // decodiertes Bit des Digital-Audio-Signals
                     DABITRDY,// Bit-Takt (1 Taktzyklus aktiv)
                     PREAMB,  // Praeambel erkannt
                     LFTCH,   // Datenblock ist fuer den linken Kanal
                     RGTCH,   // Datenblock ist fuer den rechten Kanal
                              // LFTCH u. RGTCH bleiben bis zu Beginn
                              // des naechsten Datenblocks unveraendert
                     SEQBGN;  // Datenblock ist 1. einer Sequenz; bleibt nur
                              // fuer die Dauer der Praeambelerkennung gesetzt

   reg               LFTCH,
                     RGTCH,
                     SEQBGN;

   reg      [7:0]    SR;           // Schieberegister fuer die Aufnahme von
                                  // Praeambel- und Datencodes
   reg               BITVALID,     // DABIT ist gueltig
                     BITVALID_OLD; // DABIT war im letzten Takt  schon gueltig

   wire              M,       // M-Praeambel erkannt
                     W,       // W-        "
                     B;       // B-        "


   // synopsys translate_off
   initial begin              // FPGA-Initialisierung durch GR
      SR[7:0] = 0;
      LFTCH = 0;
      RGTCH = 0;
      SEQBGN = 0;
      BITVALID = 0;
```

Listing B.5: Digital-Audio-Receiver in RTL-Verilog nach [Friedr98]

```
      BITVALID_OLD = 0;
   end
   // synopsys translate_on


   // Biphasendecodierung durch XOR der beiden Codebits
   assign DABIT = (SR[0] ^ SR[1]);

   // Datentakt aus BITVALID durch Verkuerzung auf einen aktiven Takt
    assign DABITRDY = (BITVALID & ~BITVALID_OLD);

   // Praeambelerkennung
   assign M = ((SR == 8'b11100010) || (SR == 8'b00011101));
   assign W = ((SR == 8'b11100100) || (SR == 8'b00011011));
   assign B = ((SR == 8'b11101000) || (SR == 8'b00010111));
   assign PREAMB = M | W | B;


   always @(posedge CLK) begin

      if (PREAMB) begin       // Praeambelinformationsbits setzen
         RGTCH <= W;
         LFTCH <= ~W;
         SEQBGN <= B;
         BITVALID <= 0;       // Bittakt synchronisieren
      end
      else begin
         SEQBGN <= 0;
         if (CBITRDY) BITVALID <= ~BITVALID;
         // Bittakt-Erzeugung durch Halbierung des Codebit-Taktes
      end

      if (CBITRDY) SR[7:0] <= { SR[6:0], CODEBIT };   // Codebits nachschieben
      BITVALID_OLD <= BITVALID;  // um 1 Takt verz. BITVALID-Signal
   end

endmodule
// ---------------------------------------------------------------------------



module bipdec (CLK, IECIN, BIT, BITRDY, PREAMB, LFTCH, RGTCH, SEQBGN);
// ---------------------------------------------------------------------------
// Zusammenfassung der Synchronisation und der Decodierung des
// Digital-Audio-Signals
// ---------------------------------------------------------------------------

   input             CLK,     // Takteingang
                     IECIN;   // Digital-Audio-Signal des CD-Players
   output            BIT,     // decodiertes Datenbit
                     BITRDY,  // Datenbit-Takt (1 Taktzyklus aktiv)
                     PREAMB,  // Praeambel erkannt
                     LFTCH,   // Datenblock ist fuer den linken Kanal
                     RGTCH,   // Datenblock ist fuer den rechten Kanal
                     SEQBGN;  // 1. Datenblock einer neuen Sequenz

   wire              CODEBIT, // Biphasen-Codebit
                     CBITRDY; // Codebit-Takt

   // Modul-Instanzen

   sync   SYNC   (CLK, IECIN, CODEBIT, CBITRDY);
   decode DECODE (CLK, CODEBIT , CBITRDY, BIT, BITRDY, PREAMB, LFTCH, RGTCH,
                  SEQBGN);

endmodule
// ---------------------------------------------------------------------------



module shiftin (CLK,BIT,BITRDY,PREAMB,DATABLK,DATARDY,PARITY);
// ---------------------------------------------------------------------------
// Schieberegister zum Einlesen eines kompletten Datenblocks des DA-Signals
//
```

Listing B.5: Digital-Audio-Receiver in RTL-Verilog nach [Friedr98]

```
// Liest einen gesamten Datenblock seriell ein, um ihn dann parallel
// weiterzugeben. Erkennt Pruefsummenfehler im Datenblock.
// ------------------------------------------------------------------------

   input             CLK,        // Takteingang
                     BIT,        // serieller Dateneingang
                     BITRDY,     // Datentakt
                     PREAMB;     // neuer Datenblock beginnt
   output   [27:0]   DATABLK;    // parallele Ausgabe der Daten
   output            DATARDY,    // Datenblock ist komplett eingelesen
                                 // (1 Taktzyklus aktiv)
                     PARITY;     // Paritaet; zeigt bei DATARDY = 1 einen
                                 // Pruefsummenfehler an

   reg               PARITY;

   reg      [28:0]   SR;         // Schieberegister zur Aufnahme der Daten
   reg               SRFULLN_OLD;

   wire              SRFULL;     // Datenblock ist komplett
   wire              SRFULL_OLD; // Datenblock war im letzten Takt
                                 // schon komplett


   // synopsys translate_off
   initial begin                 // FPGA-Initialisierung durch GR
      PARITY = 0;
      SRFULLN_OLD = 0;
      SR = 0;
   end
   // synopsys translate_on


   // die Daten liegen in den unteren 28 Bit des Schieberegisters
   assign DATABLK = SR[27:0];

   // das MSB des Schieberegisters dient zur Erkennung, ob 28-Bit gelesen sind
   assign SRFULL = ~SR[28];
   assign SRFULL_OLD = ~SRFULLN_OLD;         // Initialisierung mit 1, ...

   // ... um zu verhindern, dass DATARDY bei der Initialisierung gesetzt wird
   assign DATARDY = (SRFULL & ~SRFULL_OLD);  // nur einen Taktzyklus lang


   always @(posedge CLK) begin

     if (PREAMB) begin         // Beginn eines neuen Datenblocks:
        SR <= 29'b11111111111111111111111111110;
        // Zaehler: 28 Bit-Takte bis 0 ins MSB geschoben wird
        PARITY <= 0;           // Pruefsumme ruecksetzen
     end
     else if (BITRDY && ~SRFULL) begin
        // solange keine 28 Bit im Schieberegister:
        SR[28:0] <= { SR[27:0], BIT };      // neues Bit einschieben und
        PARITY <= (PARITY ^ BIT);           // Pruefsumme bilden
     end

     SRFULLN_OLD <= ~SRFULL;
     // Ein um 1 Takt verzoegertes SR-voll-Signal zur Erzeugung des fuer
     // einen Taktzyklus gesetzten DATARDY
   end

endmodule
// ------------------------------------------------------------------------



module shoreg (CLK, LFTCH, RGTCH, AMPLITUDE, LOAD, SHIFT, OUT);
// ------------------------------------------------------------------------
// Schieberegister zur Ansteuerung des DAC, synchr. parallel ladbar
// ------------------------------------------------------------------------

   input             CLK,        // Takteingang
                     LFTCH,      // linken Kanal neu setzen
```

Listing B.5: Digital-Audio-Receiver in RTL-Verilog nach [Friedr98]

```
                     RGTCH,      // rechten Kanal neu setzen
                     LOAD,       // Register par. laden (synchron)
                     SHIFT;      // Schiebetakt
   input    [11:0]   AMPLITUDE;  // 12-Bit-Amplitude aus dem Datenblock des
                                 // DA-Signals (=DATABLK[15:4])
   output            OUT;        // serieller Ausgang (=DADAT)

   reg      [15:0]   SR;         // Schieberegister


   // synopsys translate_off
   initial begin                 // FPGA-Initialisierung durch GR
      SR = 0;
   end
   // synopsys translate_on


   assign OUT = SR[0];           // LSB zuerst ausgeben


   always @(posedge CLK) begin

      if (LOAD) begin            // paralleles Laden des SR
         SR[0] <= 1;
         SR[1] <= RGTCH;
         SR[2] <= LFTCH;
         SR[3] <= 0;
         SR[4] <= ~AMPLITUDE[0];
         SR[15:5] <= AMPLITUDE[11:1];
         // vorzeichenbehaftete Digital-Audio-Werte durch Negation des MSB
         // in den posisitiven Bereich verschieben
      end
      else if (SHIFT)
         SR[15:0] <= { 1'b0, SR[15:1] };  // naechstes Bit ausgeben
   end

endmodule
// ---------------------------------------------------------------------------



module shoctrl (CLK, STRTOUT, SHIFTCLK, DACSN, DALDN);
// ---------------------------------------------------------------------------
// Ansteuerung des Schieberegisters (Schiebetakt) und
// des Digital-Analog-Konverters (Chipselect und Load-Signal)
// ---------------------------------------------------------------------------

   input             CLK,        // Takteingang
                     STRTOUT;    // Ausgabe beginnen
   output            SHIFTCLK,   // Schiebetakt = halber Systemtakt,
                                 // also 8 MHz
                     DACSN,      // Chipselect des DAC, active low
                     DALDN;      // Load Data des DAC, active low

   reg      [4:0]    COUNT;      // Zaehler bis 16,
                                 // plus eine Stufe zur Takthalbierung
   reg               DACS,       // active high
                     DALD;       // active high


   // synopsys translate_off
   initial begin                 // FPGA-Initialisierung durch GR
      COUNT = 0;
      DACS = 0;
      DALD = 0;
   end
   // synopsys translate_on


   assign DACSN = ~DACS;         // dadurch Initialisierung mit 1
   assign DALDN = ~DALD;         // s.o.

   // Erzeugung des Schiebetaktes
   assign SHIFTCLK = COUNT[0];   // Takt durchlaeuft eine Zaehlerstufe
```

Listing B.5: Digital-Audio-Receiver in RTL-Verilog nach [Friedr98]

```
   always @(posedge CLK) begin

      if (STRTOUT) begin
         DACS <= 1;              // Ausgabe starten
         COUNT <= 0;             // Zaehler Ruecksetzen, SHIFTCLK wird 0 im
                                 // naechsten Takt, kann also hierdurch
                                 // 2 statt 1 Takte auf 0 liegen
      end
      else begin
         if (COUNT == 31) begin  // 16 Bit geschoben,
            DACS <= 0;           // Ausgabe beenden
            if (DACS) DALD <= 1; // Analogwandlung starten
         end
         COUNT <= COUNT + 1;
      end

      if (DALD) DALD <= 0;       // DALD nach einem Takt wieder ruecksetzen
   end

endmodule
// ---------------------------------------------------------------------------



module dacout (CLK, STRTOUT, LFTCH, RGTCH, AMPLITUDE,
               DACLK, DADAT, DACSN, DALDN);
// ---------------------------------------------------------------------------
// Ausgabe des Audio-Wertes des entsprechenden Kanals an den
// Digital-Analog-Wandler
// ---------------------------------------------------------------------------

   input             CLK,        // Takteingang
                     STRTOUT,    // startet die Ausgabe an den DAC
                     LFTCH,      // linken Kanal neu setzen
                     RGTCH;      // rechten Kanal neu setzen
   input    [11:0]   AMPLITUDE;  // Audio-Amplitude,
                                 // 'falsche Wertigkeit': MSB in Bit 0
   output            DACLK,      // Eingaenge des DAC: Taktsignal
                     DADAT,      //   serieller Dateneingang
                     DACSN,      //   Chip Select
                     DALDN;      //   Load

   // Modul-Instanzen

   shoctrl SHOCTRL (CLK, STRTOUT, DACLK, DACSN, DALDN);
   shoreg  SHOREG  (CLK, LFTCH, RGTCH, AMPLITUDE, STRTOUT, DACLK, DADAT);

endmodule
// ---------------------------------------------------------------------------



module streg (CLK, SEQBGN, STATUSBIT, STBITRDY, VALID, COPYR, COPYG);
// ---------------------------------------------------------------------------
// Register zur Aufnahme der Statusdaten des Digital-Audio-Signals
//
// Es werden von den 192 moeglichen nur die Statusbits 1,2,3 und 16
// aufgefangen und ausgewertet, da nur diese zur Erzeugung der
// Zustandssignale VALID, COPYR und COPYG benoetigt werden.
// ---------------------------------------------------------------------------

   input             CLK,        // Takteingang
                     SEQBGN,     // Beginn einer neuen Sequenz von Statusbits
                     STATUSBIT,  // aktuelles Statusbit,
                     STBITRDY;   // neues Statusbit liegt an (1 Takt)
   output            VALID,      // gueltiges SPDIF-Audio-Signal, wenn 1
                     COPYR,      // Kopierschutz aktiv, wenn 1
                     COPYG;      // Kopiergeneration, Kopie wenn 1

   reg               COPYR,
                     COPYG;
```

Listing B.5: Digital-Audio-Receiver in RTL-Verilog nach [Friedr98]

```
   reg      [7:0]    COUNT;      // 8-Bit-Zaehler, da 192 mögliche Statusbits
   reg               SPDIF,      // Uebertragungsmodus ist SPDIF
                     AUDIOVALID; // Audio-Signal ist gueltig


   // synopsys translate_off
   initial begin
      COUNT = 0;                 // FPGA-Initialisierung durch GR
      SPDIF = 0;
      AUDIOVALID = 0;
      COPYR = 0;
      COPYG = 0;
   end
   // synopsys translate_on


   assign VALID = SPDIF & AUDIOVALID;  // gueltig und SPDIF


   always @(posedge CLK) begin

      if (SEQBGN)                // zu Beginn der Sequenz den
         COUNT <= 0;             // Zaehler ruecksetzen

      if (STBITRDY) begin        // neues Statusbit liegt an,

         if (COUNT == 0)
            SPDIF <= ~STATUSBIT;

         if (COUNT == 1)
            AUDIOVALID <= ~STATUSBIT;

         if ((COUNT == 2) && VALID)
            COPYR <= ~STATUSBIT;

         if ((COUNT == 15) && VALID)
            COPYG <= STATUSBIT & COPYR;

         COUNT <= COUNT + 1;     // Statusbit-Zaehler erhoehen
      end
   end

endmodule
// ---------------------------------------------------------------------------



module ctrl (CLK, DATARDY, PARITY, INTERPOL, VALID, SEQBGN, LFTCH, STBITRDY,
             DATAOUT);
// ---------------------------------------------------------------------------
// Lesen von Statusbits steuern und die Ausgabe bei fehlerhaften oder
// interpolierten Audio-Daten unterdruecken.
//
// Die Statusbits sind redundant jeweils in den Daten fuer den linken und fuer
// den rechten Kanal vorhanden. Sie werden nur aus den Daten fuer den linken
// Kanal gelesen. Auch wenn dies fehlschlaegt, werden die Daten fuer den
// rechten Kanal nicht beruecksichtigt.
// ---------------------------------------------------------------------------

   input             CLK,        // Takteingang
                     DATARDY,    // neuer Datenblock steht bereit
                     PARITY,     // Paritaetsfehler (bei DATARDY = 1 gueltig)
                     INTERPOL,   // Audio-Daten sind interpoliert
                     VALID,      // gueltiges SPDIF-Audio-Signal
                     SEQBGN,     // neue Sequenz von Datenbloecken/Statusbits
                     LFTCH;      // Daten sind fuer den linken Kanal;
   output            STBITRDY,   // Statusbit einlesen
                     DATAOUT;    // Ausgabe der Daten und DA-Wandlung starten

   reg               STBITRDY;

   reg               STERRN,     // Fehler beim Lesen von Statusbits innerhalb
                                 // einer Sequenz, active low
```

Listing B.5: Digital-Audio-Receiver in RTL-Verilog nach [Friedr98]

```
                     DATAVALID1, // Audio-Daten fehlerfrei eingelesen und
                                 // nicht interpoliert
                     DATAVALID2, // um einen Takt verzoegertes DATAVALID1, zur
                                 // Synchronisation mit dem VALID-Signal
                     DATAOUT;    // Ausgabe der Audio-Daten veranlassen

   wire              STERR;      // Statusbit-Lesefehler innerhalb einer Seq.


   // synopsys translate_off
   initial begin
      STERRN = 0;                // FPGA-Initialisierung durch GR
      DATAVALID1 = 0;
      DATAVALID2 = 0;
      DATAOUT = 0;
      STBITRDY = 0;
   end
   // synopsys translate_on


   assign STERR = ~STERRN;       // so Initialisierung mit 1 ermoeglicht


   always @(posedge CLK) begin

      if (SEQBGN)                // Am Anfang der Datenblock-Sequenz
         STERRN <= 1;            // Lesefehler ruecksetzen

      else if (DATARDY && PARITY && LFTCH)
         STERRN <= 0;            // Paritaetsfehler, keine weiteren
                                 // Statusbits dieser Sequenz akzeptieren

      // wenn kein Paritaetsfehler innerhalb dieser Sequenz aufgetreten ist,
      // naechstes Statusbit einlesen
      STBITRDY <= DATARDY & ~PARITY & ~STERR & LFTCH;

      // DATARDY und PARITY mit VALID synchronisieren, da VALID 2 Takte
      // spaeter gueltig wird
      DATAVALID1 <= DATARDY & ~PARITY & ~INTERPOL;
      DATAVALID2 <= DATAVALID1;

      // Ausgabe starten, wenn es sich um ein gueltiges, nicht interpoliertes
      // SPDIF-Audio-Signal handelt, das fehlerfrei gelesen werden konnte
      DATAOUT <= DATAVALID2 & VALID;
   end

endmodule
// ---------------------------------------------------------------------------



module control (CLK, DATARDY, PARITY, INTERPOL, SEQBGN, LFTCH, STBIT, STRTOUT,
                VALID, COPYR, COPYG);
// ---------------------------------------------------------------------------
// Statusbits lesen und auswerten. Aufgrund des Status, Paritaet und
// Interpolationsbit die Ausgabe der Audio-Daten starten, bzw unterdruecken.
// ---------------------------------------------------------------------------

   input             CLK,        // Takteingang
                     DATARDY,    // neuer datenblock steht bereit
                     PARITY,     // Paritaetsfehler
                     INTERPOL,   // Audio-daten sind interpoliert
                     SEQBGN,     // Beginn einer Datenblock-Sequenz
                     LFTCH,      // Daten sind fuer den linken Audio-Kanal
                     STBIT;      // Statusbit des Aktuellen Datenblocks
   output            STRTOUT,    // Ausgabe der Audio-Daten starten
                     VALID,      // Audio-Daten sind gueltig und SPDIF
                     COPYR,      // 1 bei kopiergeschuetzten Daten
                     COPYG;      // Kopiergeneration: 1 bei Kopie

   wire              STBITRDY;   // neues Statusbit steht bereit

   // Modul-Instanzen
```

Listing B.5: Digital-Audio-Receiver in RTL-Verilog nach [Friedr98]

```
   ctrl  CTRL  (CLK, DATARDY, PARITY, INTERPOL, VALID, SEQBGN, LFTCH,
                STBITRDY, STRTOUT);
   streg STREG (CLK, SEQBGN, STBIT, STBITRDY, VALID, COPYR, COPYG);

endmodule
// ------------------------------------------------------------------------
```

Listing B.5: Digital-Audio-Receiver in RTL-Verilog nach [Friedr98]

B.2.2 High-Level-Synthese

In Listing B.6 ist das High-Level-Modell des Digital-Audio-Receivers entsprechend [Friedr98] dokumentiert.

```
// ------------------------------------------------------------------------
// 'Digital-Audio-Receiver'
// Akustische Wiedergabe eines digitalen Audio-Signals eines Musik-CD-Players
// ueber einen (externen) Digital-Analog-Konverter.
//
// Nach der Aufgabenstellung des VLSI-Entwurfspraktikums im
// Sommersemester 1997
//
// Verhaltensbeschreibung zur High-Level-Synthese mit Synopsys Behavioral
// Compiler.
//
// Autor: Arne Friedrichs, 12/97
// ------------------------------------------------------------------------



module dar (CLKIN, IECIN, DADAT, DACLK, DACSN, DALDN, VALID, COPYR, COPYG,
            NRST);
// ------------------------------------------------------------------------
// Top-Level-Modul des DAR, nur Modulinstanzen
//
// NRST     asynchroner Reset, active low
// CLKIN    Taktsignal (16MHz zwingend)
// IECIN    Digital-Audio-Signal des CD-Players (0V/5V-Uebertragungspegel)
// Schnittstelle zum DAC AD8522:
// DACLK    Taktsignal fuer DAC: 1/2 CLKIN = 8 MHz
// DADAT    Serieller Dateneingang des DAC
// DACSN    Freigabe fuer das Eingangsschieberegister des DAC, active low
// DALDN    Freigabesignal fuer DA-Wandlung im DAC, active low
// Weitere Ausgangssignale:
// VALID    zeigt gueltiges SPDIF-Signal an
// COPYR    zeigt kopiergeschuetzte Daten an
// COPYG    Kopiergeneration der Daten (Original/Kopie)
// ------------------------------------------------------------------------

   input             CLKIN,   // Takteingang
                     IECIN,   // Digital-Audio-Signal des CD-Players
                     NRST;    // Reset-Signal; nicht modelliert !
   output            DADAT,   // serielle Daten an den DAC
                     DACLK,   // Ausgabetakt fuer DAC
                     DACSN,   // Chip Select des DAC, active low
                     DALDN,   // Load-Signal des DAC, active low
                     VALID,   // 1, wenn gueltiges SPDIF-Signal an IECIN
                     COPYR,   // 1 bei kopiergeschuetzten Daten
                     COPYG;   // 0 = 0riginal, 1 = Kopie

   wire              DABIT,   // decodiertes Datenbit des Dig.-Audio-Sign.
                     DABITRDY,// Datenbit-Takt, jeweils 1 Taktzyklus aktiv
                     PREAMB,  // Praeambel erkannt
                     LFTCH,   // Daten sind fuer den linken Kanal
                     RGTCH,   // Daten sind fuer den rechten Kanal
                     SEQBGN,  // Sequenz von Datenbloecken beginnt
                     INTERPOL,// zeigt interpolierte Audio-Daten an
                     STATUSBIT,  // Statusbit im aktuellen Datenblock
                     DATARDY, // gesamter Datenblock wurde gelesen
                     PARITY;  // Paritaetspruefung des Datenblocks
   wire     [11:0]   AMPLITUDE;  // digitaler Wert der Signalamplitude
```

Listing B.6: Digital-Audio-Receiver in High-Level-Verilog nach [Friedr98]

```
   // Modul-Instanzen

   bipdec BIPDEC (CLKIN, IECIN, DABIT, DABITRDY, PREAMB,
                  LFTCH, RGTCH, SEQBGN, NRST);

   shiftin SHIFTIN (CLKIN, DABIT, DABITRDY, PREAMB, AMPLITUDE,
                    INTERPOL, STATUSBIT, DATARDY, PARITY, NRST);

   dacout DACOUT (CLKIN, SEQBGN, LFTCH, RGTCH, AMPLITUDE, INTERPOL,
                  STATUSBIT, DATARDY, PARITY, DACLK, DADAT, DACSN,
                  DALDN, VALID, COPYR, COPYG, NRST);

endmodule
// ------------------------------------------------------------------------



module bipdec (CLKIN, IECIN, DABIT, DABITRDY, PREAMB, LFTCH, RGTCH, SEQBGN,
               NRST);
// ------------------------------------------------------------------------
// Synchronisation und Decodierung des Digital-Audio-Signals
//
// Bei der geforderten Taktfrequenz von 16 MHz haben 3 Biphasen-Codebit eine
// 'Laenge' von 8 bis 9 Taktzyklen. Das Biphasensignal wechselt nach
// spaetestens 3 Bits (im Falle einer Praeambel) den Pegel.
// Nach jeweils 3 Takten wird ein Codebit abgetastet, im Falle einer
// Flanke im Biphasensignal entsprechend frueher.
// Ein Datenbit wird durch XOR von zwei Codebits gewonnen.
// In einem 8-Bit Schieberegister wird das eingelesene Biphasensignal mit
// den moeglichen Praeambelcodes verglichen.
// ------------------------------------------------------------------------

   input             CLKIN,   // Takteingang
                     IECIN,   // Digital-Audio-Signal
                     NRST;    // asynchroner Reset; nicht modelliert !
   output            DABIT,   // dekodiertes Bit des Digital-Audio-Signals
                     DABITRDY,// Bit-Takt, jeweils 1 Taktzyklus gesetzt
                     PREAMB,  // Praeambel erkannt
                     LFTCH,   // Datenblock ist fuer den linken Kanal
                     RGTCH,   // Datenblock ist fuer den rechten Kanal
                     SEQBGN;  // Datenblock ist 1. einer Sequenz


   reg               DABIT,
                     DABITRDY,
                     PREAMB,
                     LFTCH,
                     RGTCH,
                     SEQBGN,

                     DABITCLK,// zur Erzeugung des Datenbittaktes DABITRDY
                     M,W,B;   // Praeambeltypen

   reg      [1:0]    SYNCSR,  // Schieberegister zur Biphasen-Dekodierung
                     CYCLECNT;// Abtastzaehler
   reg      [7:0]    CODEBITSR;  // Schieberegister fuer Praeambelvergleich


   // M-Praeambel ?
   function MPREAMB;
      input [7:0] BYTE;
      MPREAMB = ((BYTE == 8'b11100010) || (BYTE == 8'b00011101));
   endfunction

   // W-Praeambel ?
   function WPREAMB;
      input [7:0] BYTE;
      WPREAMB = ((BYTE == 8'b11100100) || (BYTE == 8'b00011011));
   endfunction

   // B-Praeambel ?
   function BPREAMB;
      input [7:0] BYTE;
```

Listing B.6: Digital-Audio-Receiver in High-Level-Verilog nach [Friedr98]

```
      BPREAMB = ((BYTE == 8'b11101000) || (BYTE == 8'b00010111));
   endfunction

   // synopsys translate_off
   initial begin
      DABIT = 0;
      DABITRDY = 0;
      PREAMB = 0;
      LFTCH = 0;
      RGTCH = 0;
      SEQBGN = 0;
      DABITCLK = 0;
      M = 0;
      W = 0;
      B = 0;
      SYNCSR = 2'b0;
      CYCLECNT = 2'b0;
      CODEBITSR = 8'b0;
   end
   // synopsys translate_on

   always begin : BIPDEC

      // Initialisierung (Register, die nur beschrieben, jedoch nicht
      // gelesen werden und mit Null zu initialisieren sind, sind hier
      // nicht aufgefuehrt, da der Global Reset des FPGAs alle Register
      // mit Null initalisiert)
      SYNCSR = 2'b00;
      CYCLECNT = 2'b00;
      DABITCLK = 0;
      CODEBITSR[7:0] = 8'b10101010; // um zu Beginn falsche Preambelerkennung
                                    // zu verhindern
      @(posedge CLKIN);

      forever begin : MAIN_LOOP

         // Neues Codebit nach 3 Takten oder Flanke im Signal
         if ((SYNCSR[1] != SYNCSR[0]) || (CYCLECNT == 2)) begin

            CYCLECNT = 0;           // Abtastzaehler ruecksetzen

            // (aelteres) Codebit ins Schieberegister aufnehmen und auf
            // Praeambelcode ueberpruefen
            CODEBITSR[7:0] = { CODEBITSR[6:0], SYNCSR[1] };
            M = MPREAMB(CODEBITSR);
            W = WPREAMB(CODEBITSR);
            B = BPREAMB(CODEBITSR);

            if (M | W | B) begin
               // Praeambel erkannt: Praeambelinformationen lesen und
               // Datenbittakt synchronisieren
               DABITCLK = 1;
               RGTCH <= WPREAMB(CODEBITSR);
               LFTCH <= ~WPREAMB(CODEBITSR);
               SEQBGN <= BPREAMB(CODEBITSR);
               PREAMB <= 1;
            end
            else begin
               // keine Praeambel: Takt negieren und DABITRDY fuer
               // einen Takt auf 1 setzen, falls DABITCLK = 1
               DABITCLK = ~DABITCLK;
               DABITRDY <= DABITCLK;
               PREAMB <= 0;
            end

            // Biphasendekodierung durch XOR der beiden Codebits
            DABIT <= CODEBITSR[1] ^ CODEBITSR[0];

         end
         else begin
            // kein neues Codebit gelesen, also Abtastzaehler erhoehen
            CYCLECNT = CYCLECNT + 1;
            DABITRDY <= 0;          // nur 1 Taktzyklus auf 1
         end
```

Listing B.6: Digital-Audio-Receiver in High-Level-Verilog nach [Friedr98]

```
         // jedesmal den neuen Wert von IECIN in SYNCSR schieben
         SYNCSR[1:0] = { SYNCSR[0], IECIN };

         @(posedge CLKIN);
         // die gesamte Verarbeitung muss in jedem Taktzyklus einmal
         // durchgefuehrt werden, also cycle_fixed-Scheduling !
      end
   end

endmodule
// ------------------------------------------------------------------------



module shiftin (CLKIN, DABIT, DABITRDY, PREAMB, AMPLITUDE, INTERPOL,
                STATUSBIT, DATARDY, PARITY, NRST);
// ------------------------------------------------------------------------
// serielles Einlesen eines kompletten Datenblocks des DA-Signals
//
// Hierbei werden die gelesenen Datenbits einer Paritaetspruefung unterzogen.
// Wurde der gesamte Block gelesen, werden die benoetigten Bits
// parallel ausgegeben.
// ------------------------------------------------------------------------

   input              CLKIN,      // Takteingang
                      DABIT,      // Datenbit des DA-Signals
                      DABITRDY,   // neues Datenbit liegt bereit
                      PREAMB,     // Praeambel erkannt
                      NRST;       // asynchroner Reset; nicht modelliert !
   output   [11:0]    AMPLITUDE;  // 12 relevante Bits der Audio-Amplitude
   output             INTERPOL,   // Interpolationsbit aus dem Datenblock
                      STATUSBIT,  // Statusbit des Datenblocks
                      DATARDY,    // Datenblock wurde komplett gelesen
                                  // (1 Taktzyklus aktiv)
                      PARITY;     // Paritaet; zeigt bei DATARDY = 1 einen
                                  // Pruefsummenfehler an

   reg                DATARDY,
                      PARITY;

   reg      [15:0]    SR;         // Schieberegister zur Aufnahme der Daten
                                  // von den 28 Bits des Blocks werden aber
                                  // nur die unteren 16 gespeichert
   reg      [4:0]     i;          // Zaehlvariable


   // Position der Signale im Datenblock
   assign AMPLITUDE[11:0] = SR[15:4];
   assign INTERPOL = SR[3];
   assign STATUSBIT = SR[1];

   // synopsys translate_off
   initial begin
      DATARDY = 0;
      PARITY = 0;
      SR = 16'b0;
   end
   // synopsys translate_on

   always begin : SHIFTIN

      // Zur der Initialisierung durch den Global Reset ist hier keine
      // weitere Register-Initialisierung noetig

      // Warten, bis eine Praeambel den Beginn eines neuen Datenblocks anzeigt
      while (!PREAMB) begin : WAIT_FOR_PREAMB
         @(posedge CLKIN);
      end

      PARITY <= 0;            // Paritaet ruecksetzen

      // Schleife mit 28 Durchlaeufen zum Einlesen der Datenblock-Bits
      i = 0;
```

Listing B.6: Digital-Audio-Receiver in High-Level-Verilog nach [Friedr98]

```
      begin : SHIFT_LOOP
      forever begin

         i = i + 1;
         if (i == 29) begin
            @(posedge CLKIN);
            disable SHIFT_LOOP;
         end
         else begin

            // Warten, bis neues Datenbit bereit steht
            while (!DABITRDY) begin : WAIT_FOR_DABITRDY
              @(posedge CLKIN);
            end

            SR[15:0] <= { SR[14:0], DABIT }; // neues Bit einschieben
            PARITY <= PARITY ^ DABIT;        // Paritaet generieren
            @(posedge CLKIN);
         end
      end
      end

      // Nach den Einlesen der Daten DATARDY fuer einen Taktzyklus setzen
      DATARDY <= 1;
      @(posedge CLKIN);

      DATARDY <= 0;
      @(posedge CLKIN);

   end // always
endmodule
// ---------------------------------------------------------------------------



module dacout (CLKIN, SEQBGN, LFTCH, RGTCH, AMPLITUDE, INTERPOL, STATUSBIT,
               DATARDY, PARITY, DACLK, DADAT, DACSN, DALDN,
               VALID, COPYR, COPYG, NRST);
// ---------------------------------------------------------------------------
// serielle Ansteuerung des Digital-Audio-Konverters
//
// Auswertung der Statusbits des Digital-Audio-Signals.
// Die Audiowerte werden nur ausgegeben, wenn es sich um ein gueltiges
// SPDIF-Signal handelt, die Daten nicht interpoliert sind und kein
// Paritaetsfehler beim Lesen aufgetreten ist.
// ---------------------------------------------------------------------------

   input             CLKIN,      // Takteingang
                     SEQBGN,     // zeigt den Anfang einer neuen Sequenz von
                                 // Datenbloecken an
                     LFTCH,      // Daten sind fuer den linken Stereokanal
                     RGTCH,      // Daten sind fuer den rechten Stereokanal
                     INTERPOL,   // Audio-Daten sind interpoliert
                     STATUSBIT,  // Statusbit des Blocks
                     DATARDY,    // neue Daten liegen an
                     PARITY,     // Paritaetsfehler, gueltig bei DATARDY
                     NRST;       // asynchroner Reset; nicht modelliert
   input    [11:0]   AMPLITUDE;  // hoechstwertige 12 Bit der
                                 // Audio-Amplitude aus dem Datenblock
   output            DACLK,      // Taktsignal fuer die Eingabe am DAC
                     DADAT,      // serielle Daten fuer DAC
                     DACSN,      // Chip Select des DAC, active low
                     DALDN,      // Load-Signal des DAC (DA-Wandlung starten),
                                 // active low
                     VALID,      // Audiodaten sind gueltig und SPDIF
                     COPYR,      // 1 bei kopiergeschuetzten Daten
                     COPYG;      // Kopiergeneration: 1 bei Kopie

   reg               DACLK,
                     DADAT,
                     DACSN,
                     DALDN,
                     VALID,
                     COPYR,
```

Listing B.6: Digital-Audio-Receiver in High-Level-Verilog nach [Friedr98]

```
                    COPYG;

reg      [15:0]   SR;         // Schieberegister zur seriellen Ausgabe
reg      [4:0]    STBITCNT,   // Nummer des aktuellen Statusbits
                  i;          // Zaehlvariable
reg               AUDIOVALID, // Audiosignal ist gueltig
                  SPDIF;      // Audiosignal ist Typ 'SPDIF'

// synopsys translate_off
initial begin
   DACLK = 0;
   DADAT = 0;
   DACSN = 0;
   DALDN = 0;
   VALID = 0;
   COPYR = 0;
   COPYG = 0;
   SR = 16'b0;
   STBITCNT = 5'b0;
   AUDIOVALID = 0;
   SPDIF = 0;
end
// synopsys translate_on

always begin : DACOUT

   // Initialisierung, uebrige Register werden nur durch Global Reset
   // des FPGAs initialisiert
   SPDIF = 0;
   AUDIOVALID = 0;
   STBITCNT = 16; // ungueltig; es werden nur die Bits 0 bis 15
                  // ausgewertet, bei Werten >15 bleibt der Zaehler stehen
   DACSN <= 1;
   DALDN <= 1;
   @(posedge CLKIN);

   forever begin : MAIN_LOOP

      // Warten, bis neuer Datenblock eingelesen wurde
      while (!DATARDY) begin : WAIT_FOR_DATARDY
         @(posedge CLKIN);
      end

      // erst Statusbits verarbeiten ...

      if (PARITY & LFTCH)
         STBITCNT = 16;    // Paritaetsfehler, Rest der Sequenz ignorieren
      else if (SEQBGN)
         STBITCNT = 0;     // neue Sequenz von Statusbits, Zaehler init.

      // Es werden nur die Statusbit aus den Datenbloecken fuer den
      // linken Kanal betrachtet; redundant in den Bloecken fuer rechten K.
      if (LFTCH) begin
         case (STBITCNT)   // synopsys parallel_case
            0  :            SPDIF = ~STATUSBIT;
            1  :            AUDIOVALID = ~STATUSBIT;
            2  : if (VALID) COPYR <= ~STATUSBIT;
            15 : if (VALID) COPYG <= STATUSBIT & COPYR;
         endcase

         if (!STBITCNT[4]) // STBITCOUNT < 16
            STBITCNT = STBITCNT + 1;   // Zaehler ggf. erhoehen
      end

      VALID <= SPDIF & AUDIOVALID;     // gueltige SPDIF-Audiodaten
      @(posedge CLKIN);

      // ... dann Audiodaten ausgeben

      if (VALID & ~PARITY & ~INTERPOL) begin

         // Audiodaten sind SPDIF, gueltig und nicht interpoliert
         // Daten in das interne Schieberegister uebernehmen und Amplitude
         // durch Negation des hoechstwertigen Bits (liegt in Bit 0 !)
```

Listing B.6: Digital-Audio-Receiver in High-Level-Verilog nach [Friedr98]

```
            // von der Komplement-Darstellung in positiven Bereich
            // verschieben (Addition mit binaer 10000...0)
            SR[15:0] = { AMPLITUDE[11:1], ~AMPLITUDE[0],
                         1'b0, LFTCH, RGTCH, 1'b1 };

            DADAT <= SR[0];      // erstes auszugebendes Bit
            DACSN <= 0;          // DAC anwaehlen
            DACLK <= 0;          // Eintaktungssignal ruecksetzen
            @(posedge CLKIN)

            // Schleife mit 16 Durchlaeufen zur seriellen Ausgabe der Bits
            i = 0;
            begin : SHIFT_LOOP
            forever begin

               i = i + 1;
               if (i == 16) begin : I4
                  DACLK <= 1;       // letzte steigende Flanke an DACLK
                  @(posedge CLKIN);
                  disable SHIFT_LOOP;
               end
               else begin : E4

                  // Wert im Schieberegister nach rechts schieben
                  SR[15:0] = { 1'b0, SR[15:1] };
                  DACLK <= 1;       // Daten an DADAT gueltig (steigende Fl.)
                  @(posedge CLKIN);

                  DADAT <= SR[0];   // naechtstes Bit anlegen
                  DACLK <= 0;       // usw.
                  @(posedge CLKIN);
               end
            end
            end

            DACSN <= 1;          // Chip Select zuruecknehmen
            DALDN <= 0;          // Digital-Audio-Wandlung starten
            DACLK <= 0;
            @(posedge CLKIN);
            DALDN <= 1;          // Wandlungssignal zuruecksetzen
            DACLK <= 1;
            @(posedge CLKIN);
         end
         else
            // Ausgabe unterdruecken
            @(posedge CLKIN);

      end
   end // always

endmodule
// ------------------------------------------------------------------------
```

Listing B.6: Digital-Audio-Receiver in High-Level-Verilog nach [Friedr98]

B.2.3 Controllersynthese

Dieser Abschnitt zeigt das Protocol-Compiler-Modell des Digital-Audio-Receivers in seiner Hierarchie (Bild B.2) und den Frame-Definitionen (Bild B.3 bis Bild B.5).

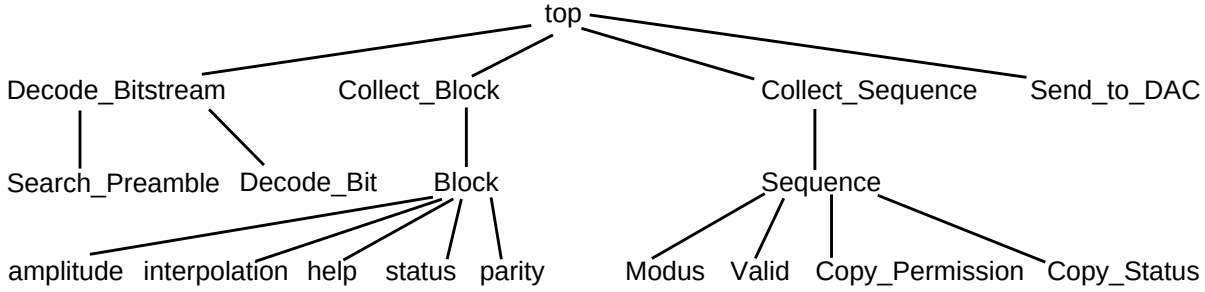


Bild B.2: Frame-Hierarchie des Digital-Audio-Receivers

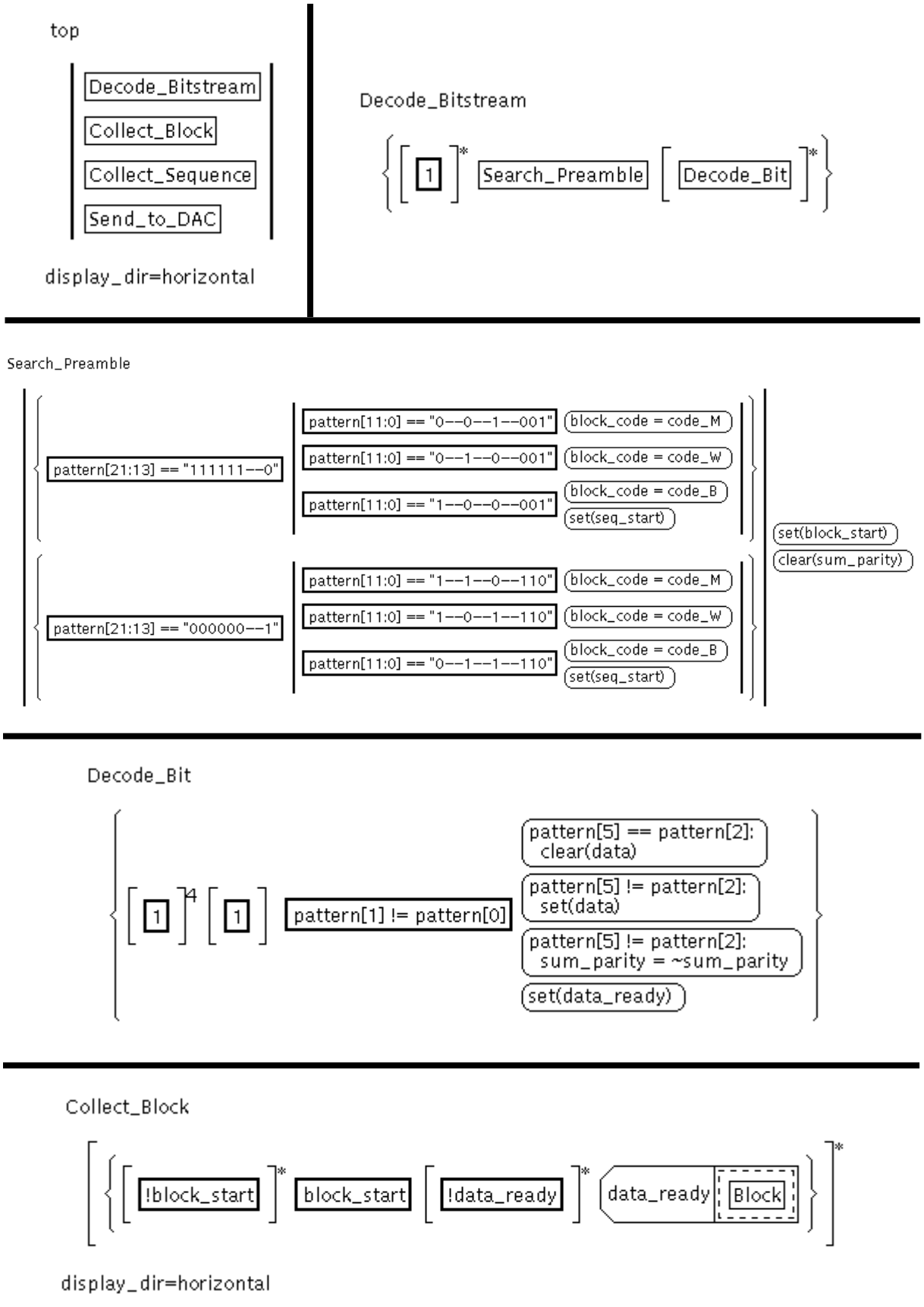


Bild B.3: Protocol-Compiler-Modell des Digital-Audio-Receivers (Teil 1)

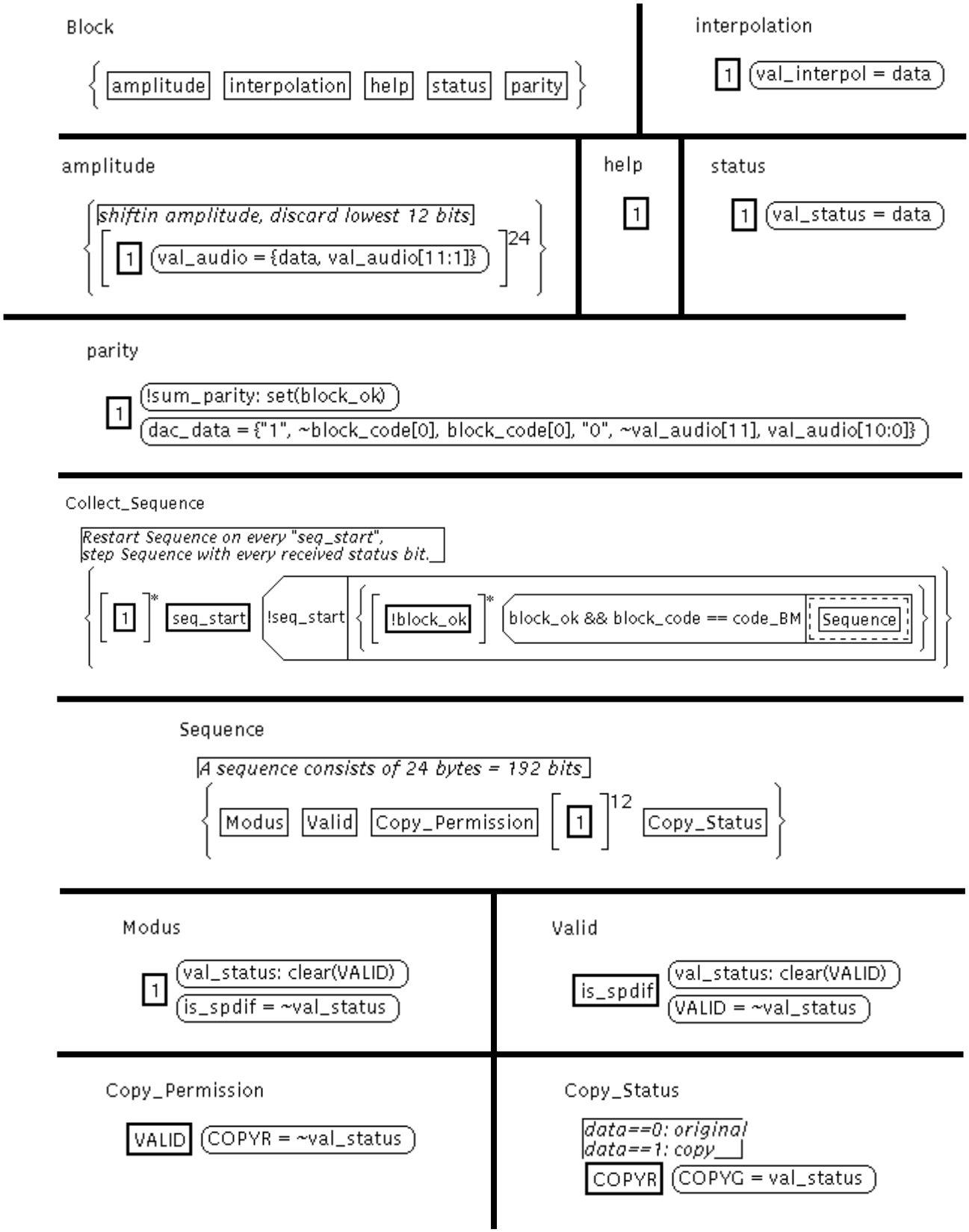
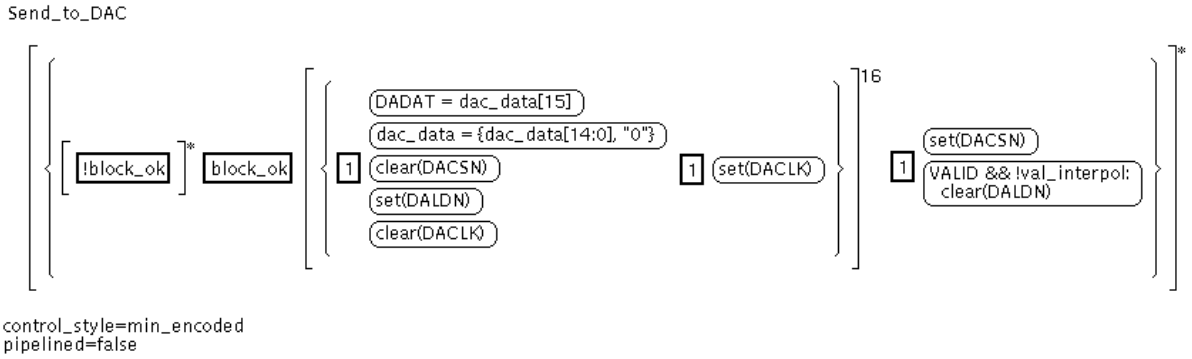


Bild B.4: Protocol-Compiler-Modell des Digital-Audio-Receivers (Teil 2)




```
Expressions:                    Ports:                  Variablen:
 code_M => "01"                  CLK, in                 block_start
 code_B => "11"                  RESET, in               pattern[21:0]
 code_W => "00"                  IECIN, in               block_code[1:0]
                                 DADAT, out              data_ready
                                 DACLK, out              data
                                 DACSN, out              val_audio[11:0]
Default-Actions:                 DALDN, out              seq_start
 clear(block_start)              VALID, out              is_spdif
 clear(block_ok)                 COPYR, out              val_status
 clear(data_ready)               COPYG, out              block_ok
 clear(seq_start)                                        sum_parity
 pattern = {pattern[20:0], IECIN}                        dac_data[15:0]
                                                         val_interpol
```

Bild B.5: Protocol-Compiler-Modell des Digital-Audio-Receivers (Teil 3)

B.3 MRISC-Prozessor

B.3.1 RTL-Synthese

Listing B.7 zeigt das RTL-Modell des MRISC-Prozessors nach [Blinze96], Listing B.8 die alternative RTL-Implementierung nach [Friedr98].

```
//------------------------------------------------------------------------------
//
//  Modellbeschreibung:   Verilog-Modell eines MRISC-Prozessors
//
//  Projektbeschreibung:  Verilog-Modellierung eines MRISC-Prozessors
//                        Synthesegeeignet fuer Synopsys
//                        VLSI-Entwurfs-Praktikum fuer SC-ASICs
//                        Technische Universitaet Braunschweig
//                        Abteilung Entwurf integrierter Schaltungen (E.I.S.)
//
//  Dateibezeichnung:     mrisc.v
//  Aktualisierungsdatum: 31. Maerz 1996
//  Implementiert von:    Peter Blinzer
//
//------------------------------------------------------------------------------
//
//  Der MRISC ist ein n-Bit Prozessor (n>4), dessen einzige Instruktion der
//   zwei-Adress-Befehl [MOVE SRCADR,DSTADR] ist. Durch die Ansprechbarkeit
//   eines n-Bit Program-Counters (PC) und einer n-Bit ALU ueber Adressen
//   der MOVE-Instruktion wird dennoch Berechnungsuniversalitaet erreicht.
//  Die Architektur basiert auf dem ultimativen RISC (URISC), welcher in
//   den SC-Praktika der Jahre 1992-1995 als Entwurfsaufgabe gestellt wurde.
//  Der MRISC wurde im Vergleich zum URISC um einige Funktionen erweitert,
//   welche den Entwurfsaufwand nur geringfuegig erhoehen und die Erstellung
//   von Programmen fuer den Prozessor wesentlich vereinfachen.
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
//  Es handelt sich hierbei um:
//   - Eine Erweiterung der ALU-Operationen (Shift, Rotate)
//   - Zwei n-Bit Register (P1, P2) fuer indirekte Adressierung
//  Die indirekte Adressierung erfolgt zur Vereinfachung des Aufbaus nur
//   auf dem externen Speicherbus des MRISC, indirekte Zugriffe auf
//   MRISC-Adressen sind daher nicht moeglich.
//  Die Adressbelegung hat sich im Vergleich zum URISC veraendert,
//   so dass die Programme von URISC und MRISC nicht binaerkompatibel sind.
//  Bei der Ausfuehrung von ALU Rechenoperationen wird wie beim URISC der
//   Akkumulator AC der ALU als Quell- und Ergebnisregister verwendet.
//  Der Wert von n wird von der Umgebung ueber ‘define WIDTH festgelegt.
//  Da der Prozessor nur einen Befehl kennt, gibt es keine Befehlscodes.
//  Die Befehlsinformation ist vollstaendig in den beiden Adressoperanden
//   enthalten, wobei die folgende Adressbelegung festgelegt wurde:
//
//   Adresse |  Bedeutung als SRCADR      | Bedeutung als DSTADR    | Bei FETCH
//   ---------------------------------------------------------------------------
//    $0000  | (DSTADR) <- PC + 4         | PC   <- (SRCADR)        | HALT
//    $0001  | (DSTADR) <- P1             | P1   <- (SRCADR)        | HALT
//    $0002  | (DSTADR) <- (P1)           | (P1) <- (SRCADR)        | HALT
//    $0003  | (DSTADR) <- P2             | P2   <- (SRCADR)        | HALT
//    $0004  | (DSTADR) <- (P2)           | (P2) <- (SRCADR)        | HALT
//    $0005  | (DSTADR) <-  AC            | AC   <- (SRCADR)        | HALT
//    $0006  | (DSTADR) <- ~AC            | AC   <- (SRCADR) + AC   | HALT
//    $0007  | (DSTADR) <- N-Flag         | AC   <- (SRCADR) - AC   | HALT
//    $0008  | (DSTADR) <- O-Flag         | AC   <- AC - (SRCADR)   | HALT
//    $0009  | (DSTADR) <- Z-Flag         | AC   <- (SRCADR) and AC | HALT
//    $000A  | (DSTADR) <- P-Flag         | AC   <- (SRCADR) or  AC | HALT
//    $000B  | (DSTADR) <- V-Flag         | AC   <- (SRCADR) xor AC | HALT
//    $000C  | (DSTADR) <- C-Flag         | AC   <- (SRCADR) << 1   | HALT
//    $000D  | (DSTADR) <- N xor V        | AC   <- (SRCADR) >> 1   | HALT
//    $000E  | (DSTADR) <- (N xor V) or Z | AC   <- (SRCADR) ror 1  | HALT
//    $000F  | (DSTADR) <- not (C or Z)   | AC   <- (SRCADR) asr 1  | HALT
//  >=$0010  | (DSTADR) <- MEM_SRCADR     | MEM_DSTADR <- (SRCADR)  | XADR
//
//   Bezeichnungen:
//     SRCADR  source address               Quelladresse
//     DSTADR  destination address          Zieladresse
//     FETCH   operand address fetch        Lesen der Operandenadresse
//     HALT    halt processor               Prozessor anhalten
//     XADR    execute address operand      Bearbeiten des Adressoperanden
//     PC      program counter              Programm-Adresse
//     P1      pointer register 1           Addressregister 1
//     P2      pointer register 2           Addressregister 2
//     AC      accumulator register         Akkumulatorregister
//     N       ALU-result is negative       ALU-Ergebnis ist negativ
//     O       ALU-result is odd            ALU-Ergebnis ist ungerade
//     Z       ALU-result is zero           ALU-Ergebnis ist Null
//     P       ALU-result is even parity    ALU-Ergebnis mit gerader Paritaet
//     V       ALU-result is overflow       ALU-Ergebnis mit Ueberlauf
//     C       ALU-result is carry          ALU-Ergebnis mit Uebertrag
//     and     logical and                  logisches und
//     or      logical or                   logisches oder
//     xor     logical exclusive or         logisches exklusiv-oder
//     <<      logical shift left           Bitverschiebung nach links
//     >>      logical shift rigth          Bitverschiebung nach rechts
//     ror     rotate right                 Bitrotation nach rechts
//     asr     arithmetic shift right       arithmetisches rechtsschieben
//     MEM_    external memory address      externe Speicheradresse
//
//   Zusatzbemerkungen zu den Flags:
//    - Fuer Uebertraege bei Addition und Subtraktion ist C gesetzt
//    - N, O, Z, B und V sind immer gesetzt, wenn die Bedingung erfuellt ist
//    - gesetzte Flags liefern 2, geloeschte 0 (...Sprungzielindizierung)
//    - N xor V  dient fuer die Realisierung von [<] und [>=] mit [-]
//    - (N xor V) or Z  dient fuer die Realisierung von [<=] und [>] mit [-]
//    - not (C or Z)  dient fuer die Realisierung von [>]
//    - durch eine Operation unbestimmte Flags werden geloescht
//
//  Allgemeines Arbeitsverfahren (Verhalten):
//    Die Bearbeitung der MOVE Instruktion erfolgt in vier sequentiellen
//     Bearbeitungsschritten, den sogenannten Mikrozyklen (micro cycles):
//     1.MC: Adressoperand bei PC in internes Adressregister ADDR laden
//     2.MC: PC <- PC+1, internes Datenregister TEMP mit MEM(ADDR) laden
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
//     3.MC: Adressoperand bei PC in internes Adressregister ADDR laden
//     4.MC: PC <- PC+1, internes Datenregister TEMP nach MEM(ADDR) schreiben
//
//-----------------------------------------------------------------------------
//
//  Prozessor-Schnittstelle:
//   ADDR[n-1:0] Adressbus     (Ausgang)
//   DATA[n-1:0] Datenbus      (bidirektional, Ausgaben nur bei CLK=0)
//   CLK         Taktsignal    (Eingang, symmetrisch, Takt beginnt mit CLK=1)
//   nRESET      RESET         (Eingang, asynchron, 0-aktiv)
//   RDnWR       Datenrichtung (Ausgang, 1=lesen, 0=schreiben)
//   HALT        HALT-Anzeige  (Ausgang, 1=Prozessor inaktiv, RD, DATA=TS)
//
//-----------------------------------------------------------------------------
//
//  Busprotokoll:
//
//               |   MC1   |   MC2   |   MC3   |   MC4   |
//                _____     _____     _____     _____     __
//       CLK   __|     |___|     |___|     |___|     |___|
//              __ _________ _________ _________ _________ __
//       ADDR  __|_________|_________|_________|_________|__
//              ___       _____     _____     _____     _____
//       DATA  ___|-----|___|----|___|----|___|--|_____|-
//                 _____________________________________     __
//       RDnWR __|                                     |___|
//
//       Ist die Quelladresse MRISC-intern, so ist DATA in MC2 irrelevant.
//       Ist die Zieladresse MRISC-intern, so ist DATA in MC4 das intern
//        geschriebene Datum und ADDR die angesprochene Adresse
//
//-----------------------------------------------------------------------------
//
//  Aufbau:
//   Der MRISC besteht aus 3 Baugruppen unterschiedlicher Komplexitaet:
//   1. ALU (Arithmetic Logic Unit)
//      Die "memory mapped" arbeitende ALU, die entsprechend der
//      Ansteuerung durch die IEU Daten liest, verarbeitet und ausgibt.
//   2. IEU (Instruction Execution Unit)
//      Das Steuerwerk des MRISC, in dem sich PC, ADDR-Register, TEMP
//      und der MC-Generator befinden und in welchem neben ADDR und
//      RDnWR die Steuersignale fuer ALU und BL erzeugt werden.
//   3. BL (Buslogic)
//      Die Ansteuerung des externen Datenbusses des MRISC, welche
//       die Pointer-Register beinhaltet und je nach Zugriffadresse
//       direkt (ADDR) oder indirekt ((Pi)) adressierte Buszugriffe
//       ausfuehrt.
//   Der Entwurf basiert auf einer synchronen Register-Transfer-Logik.
//
//-----------------------------------------------------------------------------


`define WIDTH 16


module MRISC (ADDR, HALT, RDnWR, DATA, CLK, nRESET);

  //
  // Port-Deklarationen fuer "Pins"
  //
  output [(`WIDTH-1):0] ADDR;      // Adressbus
  output                HALT,      // HALT-Anzeige
                        RDnWR;     // Datenrichtung
  inout  [(`WIDTH-1):0] DATA;      // Datenbus
  input                 CLK,       // Taktsignal
                        nRESET;    // RESET-Signal

  //
  // Wire-Deklarationen fuer "Pins"
  //
  wire   [(`WIDTH-1):0] ADDR;      // Adressbus
  wire                  HALT,      // HALT-Anzeige
                        RDnWR;     // Datenrichtung
  wire   [(`WIDTH-1):0] DATA;      // Datenbus
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
  wire                  CLK,        // Taktsignal
                        nRESET;     // RESET-Signal

  //
  // Wire-Deklarationen fuer interne Verbindungen
  //
  wire   [('WIDTH-1):0] ALU_DOUT,   // ALU-Ergebnisbus
                        BL_ADDR,    // Zugriffsadresse von IEU
                        BL_DOUT,    // BL-Ergebnisbus
                        TEMP;       // IEU-TEMP-Datenbus
  wire   [3:0]          ALU_ADDR;   // ALU-Adresse von IEU
  wire                  ALU_LOAD,   // ALU-Steuerung von IEU
                        BL_POINT;   // Anzeige eines Pointer-Zugriffs

  //
  // Instanzierung der Prozessormodule
  //
  ALU alu (ALU_DOUT, TEMP, ALU_ADDR, ALU_LOAD, CLK, nRESET);
  IEU ieu (BL_ADDR, TEMP, ALU_ADDR, ALU_LOAD, BL_POINT, HALT, RDnWR,
           ALU_DOUT, BL_DOUT, CLK, nRESET);
  BL  bl  (ADDR, BL_DOUT, DATA, BL_ADDR, TEMP, BL_POINT, RDnWR, CLK, nRESET);

  //
  // Simulation anhalten, wenn MRISC im HALT-Zustand ist
  //
  always @(posedge CLK) if (HALT == 1'b1) $stop;

endmodule


//-----------------------------------------------------------------------------
//
//  ALU des MRISC-Prozessors
//
//  Ausfuehrung arithmetischer und logischer Verknuepfungen des
//   angelegten Operanden mit dem integrierten Akkumulator
//  Eine Operation wird durch das Anlegen einer Operationsadresse und eines
//   Operanden begonnen und durch eine positive Taktflanke bei ALU_LOAD=1
//   durch Ergebnisuebernahme in Akkumulator und Flag-Register beendet.
//  Die ALU ist damit weitgehend unabhaengig von der Taktfrequenz und
//   die Verantwortung fuer die Einhaltung der Berechnungszeit liegt bei
//   der Ansteuerung, wodurch Timingoptimierungen im Verarbeitungsschema
//   der ALU-Umgebung unabhaengig von der ALU ausgefuehrt werden koennen.
//
//-----------------------------------------------------------------------------
//
//  Beschreibung der Signale und ihrer Codierung:
//
//    ALU_LOAD  Laden von Akkumulator und Flags mit berechnetem Ergebnis
//              Ist dieses Signal bei einer positiven Taktflanke gesetzt,
//               so wird das Ergebnis, das aus dem anliegenden Operanden
//               und der Operationsadresse resultiert, in den Akkumulator
//               und die Flags uebernommen und ist somit auslesbar.
//    ALU_ADDR  Adresse der auszufuehrenden ALU-Operation fuer ALU_DIN
//               0101:  Akkumulator mit Operandenwert laden
//               0110:  Akku <- Akku + Operand
//               0111:  Akku <- Operand - Akku
//               1000:  Akku <- Akku - Operand
//               1001:  Akku <- Akku and Operand
//               1010:  Akku <- Akku or  Operand
//               1011:  Akku <- Akku xor Operand
//               1100:  Akku <- Operand << 1
//               1101:  Akku <- Operand >> 1
//               1110:  Akku <- Operand ror 1
//               1111:  Akku <- Operand asr 1
//               sonst: Ergebnis nicht definiert
//               Der Wert der Adresse muss rechtzeitig vor dem Laden von
//               Akkumulator und Flags durch ALU_LOAD=1 bei einer positiven
//               Taktflanke angelegt werden, damit die ALU einschwingen kann.
//              Adresse eines Ergebnisses einer ALU-Operation fuer ALU_DOUT
//               0101:  Akkumulatorinhalt
//               0110:  invertierter Akkumulatorinhalt
//               0111:  {'b0, N, 1'b0}, "negative", Bit n-1 des Akkumulators
//               1000:  {'b0, O, 1'b0}, "odd", Bit 0 des Akkumulators
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
//               1001:  {'b0, Z, 1'b0}, "zero", NOR ueber Akkumulator[n-1:0]
//               1010:  {'b0, P, 1'b0}, "parity", XOR ueber Akkumulator[n-1:0]
//               1011:  {'b0, V, 1'b0}, "overflow", Arithmetik-Ueberlauf
//               1100:  {'b0, C, 1'b0}, "carry", Arithmetik-Uebertrag
//               1101:  {'b0, (N xor V), 1'b0}
//               1110:  {'b0, ((N xor V) or Z), 1'b0}
//               1111:  {'b0, (not (C or Z)), 1'b0}
//               sonst: nicht definiert
//    ALU_DIN   Operand fuer ALU-Operation
//               Der Wert des Operanden muss rechtzeitig vor dem Laden von
//               Akkumulator und Flags durch ALU_LOAD=1 bei einer positiven
//               Taktflanke angelegt werden, damit die ALU einschwingen kann.
//    ALU_DOUT  Ueber ALU_ADDR adressiertes ALU-Ergebnis
//              Der Wert auf diesem Bus wird erst zu Beginn desjenigen
//               Taktes gueltig, bei dessen positiver Flanke ALU_LOAD=1 ist,
//               da erst mit dieser Flanke der Akkumulator und die Flags
//               mit den Ergebniswerten geladen werden.
//    CLK       Taktsignal (symmetrisch, Takt beginnt mit CLK=1)
//    nRESET    RESET-Signal fuer die ALU-Register (0-aktiv)
//
//-----------------------------------------------------------------------------

module ALU (ALU_DOUT, ALU_DIN, ALU_ADDR, ALU_LOAD, CLK, nRESET);

  //
  // Port-Deklarationen fuer Modulschnittstelle
  //
  output [('WIDTH-1):0] ALU_DOUT;   // Ergebnis-Ausgang
  input  [('WIDTH-1):0] ALU_DIN;    // Operanden-Eingang
  input  [3:0]          ALU_ADDR;   // Operations-/Ergebnisadresse
  input                 ALU_LOAD,   // Akku und Flags laden
                        CLK,        // Taktsignal
                        nRESET;     // RESET-Signal

  //
  // Register-Deklaration fuer Modulschnittstelle
  //
  reg    [('WIDTH-1):0] ALU_DOUT;   // Ergebnis-Ausgaberegister

  //
  // Wire-Deklarationen fuer Modulschnittstelle
  //
  wire   [('WIDTH-1):0] ALU_DIN;    // Operanden-Eingang
  wire   [3:0]          ALU_ADDR;   // Operations-/Ergebnisadresse
  wire                  ALU_LOAD,   // Akku und Flags laden
                        CLK,        // Taktsignal
                        nRESET;     // RESET-Signal

  //
  // Deklarationen der ALU-Ergebnisregister
  //
  reg    [('WIDTH-1):0] ACCU;       // Akkumulatorregister
  reg                   CFLG,       // Carry-Flag-Register
                        VFLG;       // Overflow-Flag-Register

  //
  // Deklarationen von Registern fuer kombinatorische Berechnungen
  //
  reg    ['WIDTH:0]     SIGN_ACCU,  // Um ein Bit vorzeichenerweiterter ACCU
                        SIGN_DIN,   // Um ein Bit vorzeichenerweitertes DIN
                        SIGN_TEMP;  // vorzeichenerweiterte Zwischenergebnisse
  reg    [('WIDTH-1):0] CALC_ACCU;  // Berechnungsregister fuer Akkumulator
  reg                   CALC_CFLG,  // Berechnungsregister fuer Carry-Flag
                        CALC_NFLG,  // Berechnungsregister fuer Negative-Flag
                        CALC_VFLG,  // Berechnungsregister fuer Overflow-Flag
                        CALC_ZFLG;  // Berechnungsregister fuer Zero-Flag

  //
  // Akkumulator-Register und Flag-Register aktualisiersen
  //
  always @(posedge CLK or negedge nRESET) begin
    if (nRESET == 1'b0) begin
      ACCU <= 'b0;
      CFLG <= 'b0;
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
      VFLG <= 'b0;
    end
    else begin
      if (ALU_LOAD == 1'b1) begin
        ACCU <= CALC_ACCU;
        CFLG <= CALC_CFLG;
        VFLG <= CALC_VFLG;
      end
    end
  end

  //
  // Ergebnis-Ausgaberegister entsprechend ALU_ADDR aktualisieren
  //
  always @(ACCU or CFLG or VFLG or CALC_NFLG or CALC_ZFLG or ALU_ADDR) begin
    case (ALU_ADDR)
      4'b0101: ALU_DOUT =  ACCU;
      4'b0110: ALU_DOUT = ~ACCU;
      4'b0111: ALU_DOUT = {1'b0, CALC_NFLG, 1'b0};
      4'b1000: ALU_DOUT = {1'b0, ACCU[0], 1'b0};
      4'b1001: ALU_DOUT = {1'b0, CALC_ZFLG, 1'b0};
      4'b1010: ALU_DOUT = {1'b0, ^ACCU, 1'b0};
      4'b1011: ALU_DOUT = {1'b0, VFLG, 1'b0};
      4'b1100: ALU_DOUT = {1'b0, CFLG, 1'b0};
      4'b1101: ALU_DOUT = {1'b0, (CALC_NFLG ^ VFLG), 1'b0};
      4'b1110: ALU_DOUT = {1'b0, ((CALC_NFLG ^ VFLG) | CALC_ZFLG), 1'b0};
      4'b1111: ALU_DOUT = {1'b0, ~(CFLG | CALC_ZFLG), 1'b0};
      default: ALU_DOUT = 'bx;
    endcase
  end

  //
  // N-Flag und Z-Flag aus Akkumulatorinhalt berechnen
  //
  always @(ACCU) begin
    CALC_NFLG = ACCU['WIDTH-1];
    CALC_ZFLG = ~(|ACCU);
  end

  //
  // Akkumulator mit Operand verknuepfen, CALC_CFLG und CALC_VFLG berechnen
  //
  always @(ACCU or ALU_DIN or ALU_ADDR) begin
    SIGN_ACCU = {ACCU['WIDTH-1], ACCU[('WIDTH-1):0]};
    SIGN_DIN  = {ALU_DIN['WIDTH-1], ALU_DIN[('WIDTH-1):0]};
    case (ALU_ADDR)
      4'b0101: begin
                 SIGN_TEMP = SIGN_DIN;
                 CALC_CFLG = 1'b0;
                 CALC_VFLG = 1'b0;
               end
      4'b0110: begin
                 SIGN_TEMP = SIGN_ACCU + SIGN_DIN;
                 CALC_CFLG = (SIGN_TEMP ^ SIGN_ACCU ^ SIGN_DIN) >> 'WIDTH;
                 CALC_VFLG = ^SIGN_TEMP[('WIDTH):('WIDTH-1)];
               end
      4'b0111: begin
                 SIGN_TEMP = SIGN_DIN - SIGN_ACCU;
                 CALC_CFLG = ~(SIGN_TEMP ^ SIGN_DIN ^ ~SIGN_ACCU) >> 'WIDTH;
                 CALC_VFLG = ^SIGN_TEMP[('WIDTH):('WIDTH-1)];
               end
      4'b1000: begin
                 SIGN_TEMP = SIGN_ACCU - SIGN_DIN;
                 CALC_CFLG = ~(SIGN_TEMP ^ SIGN_ACCU ^ ~SIGN_DIN) >> 'WIDTH;
                 CALC_VFLG = ^SIGN_TEMP[('WIDTH):('WIDTH-1)];
               end
      4'b1001: begin
                 SIGN_TEMP = SIGN_ACCU & SIGN_DIN;
                 CALC_CFLG = 1'b0;
                 CALC_VFLG = 1'b0;
               end
      4'b1010: begin
                 SIGN_TEMP = SIGN_ACCU | SIGN_DIN;
                 CALC_CFLG = 1'b0;
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
                CALC_VFLG = 1'b0;
              end
      4'b1011: begin
                SIGN_TEMP = SIGN_ACCU ^ SIGN_DIN;
                CALC_CFLG = 1'b0;
                CALC_VFLG = 1'b0;
              end
      4'b1100: begin
                SIGN_TEMP = ALU_DIN << 1;
                CALC_CFLG = ALU_DIN['WIDTH-1];
                CALC_VFLG = 1'b0;
              end
      4'b1101: begin
                SIGN_TEMP = ALU_DIN >> 1;
                CALC_CFLG = ALU_DIN[0];
                CALC_VFLG = 1'b0;
              end
      4'b1110: begin
                SIGN_TEMP = {ALU_DIN[0], ALU_DIN} >> 1;
                CALC_CFLG = ALU_DIN[0];
                CALC_VFLG = 1'b0;
              end
      4'b1111: begin
                SIGN_TEMP = SIGN_DIN >> 1;
                CALC_CFLG = ALU_DIN[0];
                CALC_VFLG = 1'b0;
              end
      default: begin
                SIGN_TEMP = 'bx;
                CALC_CFLG = 'bx;
                CALC_VFLG = 'bx;
              end
    endcase
    CALC_ACCU = SIGN_TEMP[('WIDTH-1):0];
  end

endmodule


//-----------------------------------------------------------------------------
//
//  IEU des MRISC-Prozessors
//
//  Dies ist die zentrale Steuereinheit des MRISC, von der aus sowohl
//   ALU als auch BL mit Daten- und Kontrollsignalen versorgt werden.
//  Von der ALU und der BL bereitgestellte Ergebnisse laufen hier zusammen.
//  In der IEU liegen das PC-Register, auf das lesend und schreibend ueber
//   Adresse 0 zugegriffen werden kann, das ADDR-Register, das TEMP-Register,
//   das Mikrozyklen-Zustandsregister und kombinatorische Logik fuer die
//   Steuerung von ALU, BL und IEU-Datentransport.
//
//-----------------------------------------------------------------------------
//
//  Beschreibung der Signale und ihrer Codierung:
//    BL_ADDR:  Adresse fuer Buslogik (MC1/MC3: PC, MC2/MC4: ADDR)
//    TEMP:     Datum im TEMP-Register, von MC3 bis MC2 stabil
//    ALU_ADDR: Adresse von IEU fuer ALU, in MC4/MC1 und MC2/MC3 stabil
//    ALU_LOAD: ALU-Steuerung: berechnetes Ergebnis laden (Ende von MC4/MC1)
//    BL_POINT: Anzeige eines Pointer-Zugriffs fuer BL (1: 1 <= ADDR <= 4)
//    HALT:     HALT-Anzeige (0: MRISC arbeitet, 1: MRISC im HALT-Zustand)
//    RDnWR:    Lese-/Schreib-Steuerung fuer BL und Umgebung (1: RD, 0: WR)
//    IEU_ADIN: Dateneingang von der ALU
//    IEU_BDIN: Dateneingang von der BL
//    CLK:      Taktsignal (symmetrisch, Takt beginnt mit CLK=1)
//    nRESET    RESET-Signal fuer die IEU-Register (0-aktiv)
//
//-----------------------------------------------------------------------------

module IEU (BL_ADDR, TEMP, ALU_ADDR, ALU_LOAD, BL_POINT, HALT, RDnWR,
            IEU_ADIN, IEU_BDIN, CLK, nRESET);

  //
  // Port-Deklarationen fuer Modulschnittstelle
  //
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
output [(‘WIDTH-1):0] BL_ADDR,    // Adress-Ausgang fuer Buslogik
                      TEMP;       // Daten-Ausgang der IEU fuer ALU und BL
output [3:0]          ALU_ADDR;   // Adress-Ausgang fuer ALU
output                ALU_LOAD,   // ALU-Steuerung: Akku und Flags laden
                      BL_POINT,   // Anzeige eines Pointerzugriffs fuer BL
                      HALT,       // Ausgang fuer HALT-Anzeige
                      RDnWR;      // Ausgang fuer RDnWR-Signal
input  [(‘WIDTH-1):0] IEU_ADIN,   // ALU-Daten-Eingang
                      IEU_BDIN;   // BL-Daten-Eingang
input                 CLK,        // Taktsignal
                      nRESET;     // RESET-Signal

//
// Register-Deklarationen fuer Modulschnittstelle
//
reg    [(‘WIDTH-1):0] BL_ADDR;    // Adress-Ausgangsregister
reg                   ALU_LOAD,   // ALU-Steuerungsregister
                      BL_POINT,   // Pointerzugriffs-Anzeigeregister
                      RDnWR;      // RDnWR-Ausgangsregister

//
// Wire-Deklarationen fuer Modulschnittstelle
//
wire   [(‘WIDTH-1):0] IEU_ADIN,   // ALU-Dateneingang
                      IEU_BDIN;   // BL-Dateneingang
wire   [3:0]          ALU_ADDR;   // ALU-Adresse
wire                  CLK,        // Taktsignal
                      nRESET;     // RESET-Signal

//
// Deklarationen der IEU-Register
//
reg    [(‘WIDTH-1):0] ADDR,       // Operandenadresse
                      PC,         // Programmadresse (program counter)
                      TEMP;       // Operandenspeicher
reg    [1:0]          MCSTATE;    // Mikrozyklen-Zustandsregister
reg                   START,      // START-Zustandsregister
                      HALT,       // HALT-Zustandsregister
                      RNWGP,      // RDnWR-Clock-Gate-Register (posedge)
                      RNWGN;      // RDnWR-Clock-Gate-Register (negedge)


//
// Continuous Assignment fuer Adress-Extraktion von ALU_ADDR
//
assign ALU_ADDR = ADDR[3:0];

//
// Mikrozyklen-Zustandsregister und START-Register bearbeiten
//  00 = 1. Mikrozyklus
//  01 = 2. Mikrozyklus
//  10 = 3. Mikrozyklus
//  11 = 4. Mikrozyklus
//
always @(posedge CLK or negedge nRESET) begin
  if (nRESET == 1’b0) begin
    START <= 0;
    MCSTATE <= 0;
  end
  else begin
    if (HALT == 1’b0) begin
      if (START == 1’b0) START <= 1;
      else MCSTATE <= MCSTATE + 1;
    end
    else MCSTATE <= 0;
  end
end

//
// ADDR bei Abschluss des 1. und 3. Mikrozyklus von IEU_BDIN laden
//
always @(posedge CLK or negedge nRESET) begin
  if (nRESET == 1’b0)  ADDR <= ‘b0;
  else if (MCSTATE[0] == 1’b0) ADDR <= IEU_BDIN;
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
  end

  //
  // PC bei Abschluss des 2. und 4. Mikrozyklus aktualisieren
  //
  always @(posedge CLK or negedge nRESET) begin
    if (nRESET == 1'b0) PC <= 0;
    else begin
      if (START == 1'b0) PC <= 16'h0010;
      else begin
        if (MCSTATE == 2'b01) PC <= PC + 1;
        else if (MCSTATE == 2'b11) begin
          if (|ADDR) PC <= PC + 1;
          else PC <= TEMP;
        end
      end
    end
  end

  //
  // TEMP bei Abschluss des 2. Mikrozyklus aktualisieren
  //
  always @(posedge CLK or negedge nRESET) begin
    if (nRESET == 1'b0) TEMP <= 0;
    else begin
      if (MCSTATE == 2'b01) begin
        if (~(|ADDR))       TEMP <= PC + 4;    // PC lesen
        else if (ADDR < 5)  TEMP <= IEU_BDIN;  // Pointer oder RAM lesen
        else if (ADDR > 15) TEMP <= IEU_BDIN;  // RAM lesen
        else                TEMP <= IEU_ADIN;  // ALU lesen
      end
    end
  end

  //
  // HALT bei Abschluss von HALT-Zugriffen setzen
  //
  always @(posedge CLK or negedge nRESET) begin
    if (nRESET == 1'b0) HALT <= 0;
    else if (START == 1'b1)
           if ((MCSTATE[0] == 1'b0) && (PC < 16)) HALT <= 1;
  end

  //
  // RDnWR-Clock-Gate-Register mit positiver Flankenreaktion aktualisieren
  //
  always @(posedge CLK or negedge nRESET) begin
    if (nRESET == 1'b0) RNWGP <= 0;
    else RNWGP <= RNWGN;
  end

  //
  // RDnWR-Clock-Gate-Register mit negativer Flankenreaktion aktualisieren
  //
  always @(negedge CLK or negedge nRESET) begin
    if (nRESET == 1'b0) RNWGN <= 0;
    else RNWGN <= ~RNWGP;
  end

  //
  // IEU-Kombinatorik in Abhaengigkeit vom Mikrozyklus auswerten
  //
  always @(CLK or MCSTATE or PC or ADDR or RNWGP or RNWGN) begin
    if (MCSTATE[0] == 1'b0) BL_ADDR = PC;
    else                    BL_ADDR = ADDR;
    if (MCSTATE == 2'b11)   RDnWR = 1'b0 | ~(RNWGP ^ RNWGN);
    else                    RDnWR = 1'b1;
    if ((MCSTATE == 2'b00) && (ADDR > 4) && (ADDR < 16)) ALU_LOAD = 1'b1;
    else                                                 ALU_LOAD = 1'b0;
    if ((MCSTATE[0]== 1'b1) && (ADDR > 0) && (ADDR < 5)) BL_POINT = 1'b1;
    else                                                 BL_POINT = 1'b0;
  end

endmodule
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
//-----------------------------------------------------------------------------
//
//  Buslogic des MRISC-Prozessors
//
//  Ansteuerung des externen Datenbusses gemaess IEU-Steuerung
//
//-----------------------------------------------------------------------------
//
//  Beschreibung der Signale und ihrer Codierung:
//    ADDR:     Externer Adressbus des MRISC fuer Lese-/Schreibzugriffe
//    BL_DOUT:  Ausgangsdaten der BL fuer die IEU
//    DATA:     Externer Datenbus des MRISC fuer Lese-/Schreibzugriffe
//    BL_ADDR:  Von der IEU ausgegebene Adresse
//    BL_DIN:   Von der IEU geschriebene Daten (fuer RDnWR=0 an DATA)
//    BL_POINT: Anzeige eines Pointer-Zugriffs von der IEU (1-aktiv)
//    RDnWR:    Lese-/Schreibsteuerung der IEU (0: Schreiben, 1: Lesen)
//    CLK:      Taktsignal (symmetrisch, Takt beginnt mit CLK=1)
//    nRESET    RESET-Signal fuer die BL-Register (0-aktiv)
//
//-----------------------------------------------------------------------------

module BL(ADDR, BL_DOUT, DATA, BL_ADDR, BL_DIN, BL_POINT, RDnWR, CLK, nRESET);

  //
  // Port-Deklarationen fuer Modulschnittstelle
  //
  output [(‘WIDTH-1):0] ADDR,        // Externer Adressbus
                        BL_DOUT;     // BL-Daten fuer IEU
  inout  [(‘WIDTH-1):0] DATA;        // Externer Datenbus
  input  [(‘WIDTH-1):0] BL_ADDR,     // BL-Adresse von IEU
                        BL_DIN;      // BL-Daten von IEU
  input                 BL_POINT,    // Pointerzugriffs-Anzeige
                        RDnWR,       // Lese-/Schreibsteuerung von IEU
                        CLK,         // Taktsignal
                        nRESET;      // RESET-Signal

  //
  // Register-Deklaration fuer Modulschnittstelle
  //
  reg    [(‘WIDTH-1):0] ADDR,        // Externer Adressbus
                        BL_DOUT,     // BL-Ausgangsdaten fuer IEU
                        DATA_OUT;    // Datenbus-Treiber-Register

  //
  // Wire-Deklarationen fuer Modulschnittstelle
  //
  wire   [(‘WIDTH-1):0] DATA,        // Externer Datenbus
                        BL_ADDR,     // BL-Zugriffsadresse von IEU
                        BL_DIN;      // BL-Eingangsdaten von IEU
  wire                  RDnWR;       // Lese-/Schreibsteuerung von IEU

  //
  // Continuous Assignment fuer die Ansteuerung des bidirektionalen Datenbus
  //
  assign DATA = DATA_OUT;

  //
  // Deklaration der Pointerregister
  //
  reg    [(‘WIDTH-1):0] POINT1,      // Pointerregister 1
                        POINT2;      // Pointerregister 2

  //
  // Pointerregister aktualisieren
  //
  always @(posedge CLK or negedge nRESET) begin
    if (nRESET == 1’b0) begin
      POINT1 <= 0;
      POINT2 <= 0;
    end
    else begin
      case({RDnWR, BL_POINT, BL_ADDR[1:0]})
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
        4'b0101: POINT1 <= BL_DIN;
        4'b0111: POINT2 <= BL_DIN;
      endcase
    end
  end

  //
  // Adressbus aktualisieren
  //
  always @(BL_ADDR or BL_POINT or POINT1 or POINT2) begin
    case ({BL_POINT, BL_ADDR[1:0]})
      3'b100:  ADDR = POINT2;
      3'b110:  ADDR = POINT1;
      default: ADDR = BL_ADDR;
    endcase
  end

  //
  // Ansteuerung von DATA_OUT und BL_DOUT
  //
  always @(DATA or BL_DIN or BL_ADDR or
           BL_POINT or RDnWR or POINT1 or POINT2) begin
    casez ({RDnWR, BL_POINT, BL_ADDR[1:0]})
      4'b0???: begin
                 DATA_OUT = BL_DIN;
                 BL_DOUT  = BL_DIN;
               end
      4'b1101: begin
                 DATA_OUT = 'bz;
                 BL_DOUT  = POINT1;
               end
      4'b1111: begin
                 DATA_OUT = 'bz;
                 BL_DOUT  = POINT2;
               end
      default: begin
                 DATA_OUT = 'bz;
                 BL_DOUT  = DATA;
               end
    endcase
  end

endmodule
```

Listing B.7: MRISC-Prozessor in RTL-Verilog nach [Blinze96]

```
// ----------------------------------------------------------------------------
// 'MOVE-RISC'-Prozessor
// Einfacher RISC-Prozessor, der nur eine Instruktion, naemlich das
// Verschieben von Datenworten (MOVE) kennt.
//
// Nach der Aufgabenstellung des VLSI-Entwurfspraktikums fuer
// Semi-Custom-Chips im Sommersemester 1996
//
// Mixed-Mode-Beschreibung auf Register-Transfer-Ebene fuer eine spaetere
// RTL-Synthese mit Synopsys HDL Compiler for Verilog
//
// Autor: Arne Friedrichs, 02/98
// ----------------------------------------------------------------------------



module mrisc (CLK, nRST, WRITECLK, RnW, ADDR, DATA);
// ----------------------------------------------------------------------------
// Top-level Modul des MRISC, nur Modulinstanzen
//
// CLK      Takteingang
// nRST     0-aktiver asynchroner Reset-Eingang
// WRITECLK Maskierung von RnW, steuert Dauer und Laenge des RnW-Pulses
// RnW      Datenrichtung, 0 = schreiben
// ADDR     16-Bit-Adressbus
```

Listing B.8: MRISC-Prozessor in RTL-Verilog nach [Friedr98]

```
// DATA     bidirektionaler 16-Bit-Datenbus
// ----------------------------------------------------------------------------

   input             CLK,         // Taktsignal
                     nRST,        // asynchr. Resetsignal
                     WRITECLK;    // Steuerung des RnW-Pulses
   output            RnW;         // Schreibzugriff bei RnW == 0, wird
                                  // mit WRITECLK maskiert, um Anbindung an
                                  // asynchrone Speicher zu ermoeglichen
   output   [15:0]   ADDR;        // 16-Bit Adressbus
   inout    [15:0]   DATA;        // 16-Bit Datenbus

   wire     [15:0]   IEUDATA,     // interne Signale:
                     ALUDATA,     // Datenausgaenge der einzelnen Module
                     BLDATA,
                     ADR;         // interner Adressbus
   wire              WRITE;       // internes Schreib-Signal


   // Modulinstanzen

   ieu IEU (CLK, nRST, ALUDATA, BLDATA, ADR, IEUDATA, WRITE);
   alu ALU (CLK, nRST, ADR, IEUDATA, WRITE, ALUDATA);
   bl  BL  (CLK, nRST, WRITECLK, WRITE, ADR, IEUDATA, BLDATA, ADDR, DATA, RnW);

endmodule
// ----------------------------------------------------------------------------



module ieu (CLK, nRST, ADIN, BDIN, ADR, DOUT, WRITE);
// ----------------------------------------------------------------------------
// Instruction Execution Unit
//
// zentrales Steuerwerk des MRISC: Koordiniert die Programmabarbeitung,
// steuert Daten- und Adressbus; beeinhaltet den Programmzaehler PC
// ----------------------------------------------------------------------------

   input             CLK,         // Takteingang
                     nRST;        // asynchr. Resetsignal
   input    [15:0]   ADIN,        // Daten aus der ALU
                     BDIN;        // Daten aus der BL oder ext. Speicher
   output   [15:0]   ADR,         // Adressbus
                     DOUT;        // Datenausgang zur ALU, BL und ext. Speicher
   output            WRITE;       // Schreibzugriff bei WRITE == 1

   reg      [15:0]   OPADR,       // Zwischenspeicher fuer Quell- u. Ziel-
                                  // adresse
                     ADR,
                     DATAREG,     // Zwischenspeicher fuer zu verschiebende
                                  // Daten
                     PC;          // Programmzaehler
   reg               RESET;       // synchronisiertes Reset-Signal

   reg      [1:0]    STATE;       // Zustandsregister :

   // ein MOVE-Befehl wird in vier Schritten (Taktzyklen) ausgefuehrt:

   parameter [1:0]   fetchsrcadr = 2'h0,     // Quelladresse holen
                     readdata = 2'h1,        // Daten lesen
                     fetchdestadr = 2'h2,    // Zieladresse holen
                     writedata = 2'h3;       // Daten schreiben

   // synopsys translate_off
   initial begin                  // fuer Verilog-Simulation benoetigte
      STATE = fetchsrcadr;        // Initialisierung
   end
   // synopsys translate_on


   assign DOUT = DATAREG;         // Datum wird dauerhaft ausgegeben, ist
                                  // aber nur bei WRITE == 1 gueltig

   assign WRITE = (STATE == writedata);   // Aktiv im letzten Zyklus
```

Listing B.8: MRISC-Prozessor in RTL-Verilog nach [Friedr98]

```
// Adressbus-Logik
// Zum Lesen von Quell- oder Zieladresse wird der Programmzaehlerstand
// ausgegeben, sonst die entsprechende Operandenadresse
always @(STATE or PC or OPADR) begin
   case (STATE) // synopsys full_case parallel_case
      fetchsrcadr,
      fetchdestadr : ADR = PC;      // Adresszugriff
      readdata,
      writedata    : ADR = OPADR;   // Datenzugriff
   endcase
end


// Synchronisiertes Reset-Signal erzeugen
// RESET bleibt im Gegensatz zu nRST bis zur naechsten steigenden
// Taktflanke aktiv. Waehrend RESET bleibt das Zustandregister
// unveraendert bei fetchsrcadr (Quelladresse holen).
// So wird sichergestellt, dass dieser erste Schritt mind. einen ganzen
// Taktzyklus lang ist.
always @(posedge CLK or negedge nRST) begin
   if (nRST == 1'b0)
      RESET <= 1'b1;
   else
      RESET <= 1'b0;
end


always @(posedge CLK or posedge RESET) begin

   // Initialisierung
   if (RESET) begin
      PC <= 16'h0010;         // erste Programmadresse
      STATE <= fetchsrcadr;   // internen Ablauf ruecksetzen
    end

   // interne Ablaufsteuerung
   // Bearbeitung eines MOVE-Befehls in 4 Schritten (Taktzyklen)
   else begin
      case (STATE) // synopsys full_case parallel_case

         // Quelladresse holen
         fetchsrcadr : begin
            OPADR <= BDIN;          // BDIN, da Programm im ext. Sp. steht
            PC <= PC + 1;           // Programmzaehler erhoehen
         end

         // Daten lesen
         readdata : begin
            if (ADR == 16'h0000)    // PC lesen; mit Incr. bei
               DATAREG <= PC + 3;   // 'fetchsrcadr' ergibt sich  PC + 4
            else if ((ADR >= 16'h0005) && (ADR <16'h0010))
               DATAREG <= ADIN;     // ALU-Reg. lesen
            else
               DATAREG <= BDIN;     // sonst BL oder ext. Speicherzugriff
         end

         // Zieladresse holen
         fetchdestadr : begin
            OPADR <= BDIN;          // BDIN, da Programm im ext. Sp. steht
            PC <= PC + 1;           // Programmzaehler erhoehen
         end

         // Daten schreiben
         writedata : begin
            if (ADR == 16'h0000)
               PC <= DATAREG;       // Schreibzugriff auf Programmzaehler
         end
      endcase

      STATE <= STATE + 1;           // naechsten Schritt

      // synopsys translate_off
```

Listing B.8: MRISC-Prozessor in RTL-Verilog nach [Friedr98]

```
                                       // Programmzaehler waehrend der
         $display ("PC = %h",PC);      // Verilog-Simulation ausgeben.
         if (PC < 16 ) $stop;          // HALT bei Programmausf. im
                                       // internen Speicher
         // synopsys translate_on

      end
   end

endmodule
// ---------------------------------------------------------------------------



module alu (CLK, nRST, ADR, DIN, WRITE, DOUT);
// ---------------------------------------------------------------------------
// Arithmetic Logic Unit
//
// Funtionsblock zur Datenmanipulation,
// ueber den Speicherbereich $0000 - $000F ansprechbar.
// ---------------------------------------------------------------------------

   input             CLK,        // Takteingang
                     nRST,       // asynchr. Resetsignal
                     WRITE;      // Schreibzugriff bei WRITE == 1
   input    [15:0]   ADR;        // Adressbus
   input    [15:0]   DIN;        // Datenbus (Eingang)
   output   [15:0]   DOUT;       // Datenbus (Ausgang)

   reg      [15:0]   AC;            // Akkumulator
   reg               N,O,Z,P,V,C;   // Flags

   reg      [15:0]   DOUT,       // Datenausgang (Lesezugriff auf ALU)
                     ALUOUT;     // Berechnete Funktion, kombinatorische Aus-
   reg               NOUT,       // gaenge vor den Registern der ALU
                     OOUT,       // -> Schreibzugriff auf ALU
                     ZOUT,
                     POUT,
                     VOUT,
                     COUT,
                     C15;  // Uebertrag von Bit15 nach Bit16 bei Add./Subtr.


   always @(ADR or AC or DIN) begin
      // Berechnung der gewaehlten ALU-Operation
      case (ADR[3:0])  // synopsys full_case parallel_case
      // case ist nicht 'full', Werte bei ADR[3:0] < 5 jedoch irrelevant,
      // dadurch ist Optimierung der Logik moeglich
         4'h5: begin
            ALUOUT = DIN;                          // Identitaet
            COUT = 1'b0;
            VOUT = 1'b0;
         end
         4'h6 : begin
            { COUT, ALUOUT } = AC + DIN;           // Addition
            C15 = ALUOUT[15] ^ AC[15] ^ DIN[15];
            VOUT = (C15 ^ COUT); // Rueckrechnen des Ueberlaufs
         end
         4'h7 : begin                              // Subtraktion
            { COUT, ALUOUT } = { 1'b0, DIN } - { 1'b0, AC };
            C15 = ALUOUT[15] ^ ~AC[15] ^ DIN[15];
            VOUT = (C15 ^ ~COUT);// Rueckrechnen des Ueberlaufs
         end
         4'h8 : begin                              // Subtraktion
            { COUT, ALUOUT } = { 1'b0, AC } - { 1'b0, DIN };
            C15 = ALUOUT[15] ^ AC[15] ^ ~DIN[15];
            VOUT = (C15 ^ ~COUT);// Rueckrechnen des Ueberlaufs
         end
         4'h9 : begin
            ALUOUT = AC & DIN;                     // bitweises AND
            COUT = 1'b0;
            VOUT = 1'b0;
         end
         4'ha : begin
```

Listing B.8: MRISC-Prozessor in RTL-Verilog nach [Friedr98]

```
            ALUOUT = AC | DIN;                          // bitweises OR
            COUT = 1'b0;
            VOUT = 1'b0;
         end
         4'hb : begin
            ALUOUT = AC ^ DIN;                          // bitweises XOR
            COUT = 1'b0;
            VOUT = 1'b0;
         end
         4'hc : begin
            { COUT, ALUOUT } = { DIN[15:0], 1'b0 };     // Multiplikation mit 2
            VOUT = 1'b0;
         end
         4'hd : begin
            ALUOUT = { 1'b0, DIN[15:1] };               // Division durch 2
            COUT = DIN[0];
            VOUT = 1'b0;
         end
         4'he : begin
            ALUOUT = { DIN[0], DIN[15:1] };             // rotiere rechtsherum
            COUT = DIN[0];
            VOUT = 1'b0;
         end
         4'hf : begin
            ALUOUT = { DIN[15], DIN[15:1] };            // Vorzeichenrichtige
            COUT = DIN[0];                              // Division durch 2
            VOUT = 1'b0;
         end
      endcase

      // restliche Flags berechnen
      NOUT = ALUOUT[15];               // negative
      OOUT = ALUOUT[0];                // odd
      ZOUT = ~(|ALUOUT[15:0]);         // zero
      POUT = ^ALUOUT[15:0];            // parity
   end


   // Lesezugriff auf die ALU
   always @(ADR or AC or N or O or Z or P or V or C) begin

      case (ADR[3:0])  // synopsys full_case parallel_case
      // full_case, um Optimierung zu ermoeglichen, s.o.
         4'h5 : DOUT = AC;
         4'h6 : DOUT = ~AC;
         4'h7 : DOUT = {14'b0,N,1'b0};
         4'h8 : DOUT = {14'b0,O,1'b0};
         4'h9 : DOUT = {14'b0,Z,1'b0};
         4'ha : DOUT = {14'b0,P,1'b0};
         4'hb : DOUT = {14'b0,V,1'b0};
         4'hc : DOUT = {14'b0,C,1'b0};
         4'hd : DOUT = {14'b0,N^V,1'b0};     // <, >=   Vergleich nach Subtr.
         4'he : DOUT = {14'b0,N^V|Z,1'b0};   // <=, >
         4'hf : DOUT = {14'b0,~(C|Z),1'b0};  // >, <=
      endcase
   end


   // Register-Transfer
   always @(posedge CLK or negedge nRST) begin

      if (nRST == 1'b0) begin          // Initialisierung
         AC <= 16'b0;
         N <= 0;
         O <= 0;
         Z <= 0;
         P <= 0;
         V <= 0;
         C <= 0;
      end

      else begin
         // Schreibzugriff auf die ALU
         if ((ADR >= 16'h0005) && (ADR < 16'h0010)  && WRITE) begin
```

Listing B.8: MRISC-Prozessor in RTL-Verilog nach [Friedr98]

```
            AC <= ALUOUT;
            N <= NOUT;
            O <= OOUT;
            Z <= ZOUT;
            P <= POUT;
            V <= VOUT;
            C <= COUT;
         end
      end
   end

endmodule
// ---------------------------------------------------------------------------



module bl (CLK, nRST, WRITECLK, WRITE, ADR, DIN, DOUT, ADDR, DATA, RnW);
// ---------------------------------------------------------------------------
// Bus Logic
//
// Steuert den externen Adressbus, ggf. indirekte Adressierung ueber
// Pointerregister. Ansteuerung des bidirektionalen Datenbusses, sowie
// Erzeugung des RnW-Signals, das nur einen kurzen aktiven Puls (gemaess
// der aktiven Phase von WRITECLK) im Gegensatz zu WRITE, welches den
// gesamten Taktzyklus aktiv ist.
// ---------------------------------------------------------------------------

   input             CLK,        // Taktsignal
                     nRST,       // asynchr. Resetsignal
                     WRITECLK,   // Steuerung des RnW-Pulses
                     WRITE;      // internes Write-Signal, ganzer Taktzyklus
   input    [15:0]   ADR,        // interner Adressbus
                     DIN;        // Dateneingang, intern
   output   [15:0]   DOUT,       // Datenausgang, intern
                     ADDR;       // Adressbus extern
   output            RnW;        // externes Write-Signal, nur bei CLK == 0
   inout    [15:0]   DATA;       // bidirektionaler externer Datenbus

   reg      [15:0]   P1,         // Pointer-Register fuer
                     P2,         // indirekte Adressierung
                     ADDR,
                     DOUT,
                     TRI_DATA;   // Tri-State-Treiber fuer externen Datenbus


   assign DATA = TRI_DATA;

   assign RnW = ~(WRITE & WRITECLK);  // externes RnW erzeugen

   // Adressbus-Steuerung
   // externe Adresse berechnen ( durch evtl. indirekte Addressierung)
   always @(ADR or P1 or P2) begin
      case (ADR) // synopsys parallel_case
         16'h0002 : ADDR = P1;    // ind. Zugriff ueber P1
         16'h0004 : ADDR = P2;    // ind. Zugriff ueber P2
         default  : ADDR = ADR;
      endcase
   end


   // Datenbus-Steuerung
   // internen Datenbus nur bei RnW == 0 nach aussen durchschalten, sonst
   // hochohmig setzen
   always @(RnW or DIN) begin
      if (RnW)
         TRI_DATA = 'bz;         // Lesezugriff
      else
         TRI_DATA = DIN;         // Schreibzugriff
   end


   // Datenquelle fuer Lesezugriff auswaehlen (extern oder Pointer-Reg.)
   always @(ADR or DATA or P1 or P2) begin
      case (ADR)  // synopsys parallel_case
```

Listing B.8: MRISC-Prozessor in RTL-Verilog nach [Friedr98]

```
         16'h0001 : DOUT = P1;
         16'h0003 : DOUT = P2;
         default  : DOUT = DATA;
      endcase
   end


   // Register-Transfer
   always @(posedge CLK or negedge nRST) begin

      if (nRST == 1'b0) begin          // Initialisierung
         P1 <= 0;
         P2 <= 0;
      end

      else begin
         if (WRITE && (ADR == 16'h0001)) P1 <= DIN;
         if (WRITE && (ADR == 16'h0003)) P2 <= DIN;
      end
   end

endmodule
// ---------------------------------------------------------------------------
```

Listing B.8: MRISC-Prozessor in RTL-Verilog nach [Friedr98]

B.3.2 High-Level-Synthese

Das High-Level-Modell des MRISC-Prozessors nach [Friedr98] wird in Listing B.9 dokumentiert.

```
// ---------------------------------------------------------------------------
// 'MOVE-RISC'-Prozessor
// Einfacher RISC-Prozessor, der nur eine Instruktion, naemlich das
// Verschieben von Datenworten (MOVE) kennt.
//
// Nach der Aufgabenstellung des VLSI-Entwurfspraktikums fuer
// Semi-Custom-Chips im Sommersemester 1996
//
// Verhaltensbeschreibung fuer eine High-Level-Synthese mit
// Synopsys Behavioral Compiler
//
// Autor: Arne Friedrichs, 02/98
// ---------------------------------------------------------------------------



module mrisc (CLK, nRST, WRITECLK, RnW, ADDR, DATA);
// ---------------------------------------------------------------------------
// Top-level Modul des MRISC, nur Modulinstanzen
//
// CLK      Takteingang
// nRST     0-aktiver asynchroner Reset-Eingang
// WRITECLK Maskierung von RnW, steuert Dauer und Laenge des RnW-Pulses
// RnW      Datenrichtung, 0 = schreiben
// ADDR     16-Bit-Adressbus
// DATA     bidirektionaler 16-Bit-Datenbus
// ---------------------------------------------------------------------------

   input             CLK,        // Takteingang
                     nRST,       // asynchr. Resetsignal, nicht modelliert !
                     WRITECLK;   // Steuerung des RnW-Pulses
   inout    [15:0]   DATA;       // 16-Bit bidirektionaler Datenbus
   output            RnW;        // Schreibzugriff bei RnW == 0, wird
                                 // mit WRITECLK maskiert, um Anbindung an
                                 // asynchrone Speicher zu ermoeglichen
   output   [15:0]   ADDR;       // 16-Bit-Adressbus

   wire     [15:0]   DATAIN,     // interner unidir. Datenbus: Eingang,
                     DATAOUT;    // Ausgang

   // Modul-Instanzen
```

Listing B.9: MRISC-Prozessor in High-Level-Verilog nach [Friedr98]

```
   mrisc_core      MRISCCORE (CLK, nRST, WRITECLK, RnW, ADDR, DATAIN,DATAOUT);
   tristatedatabus TRISTDATA (RnW, DATAIN, DATAOUT, DATA);

endmodule
// ----------------------------------------------------------------------------



module tristatedatabus (RnW, DATA_READ, DATA_WRITE, DATA_BIDIRECT);
// ----------------------------------------------------------------------------
// 16-Bit Tri-State-Treiber zur Ansteuerung des externen bidirektionalen
// Datenbusses.
// Dieser muss als eigenstaendiges Modul auf RTL-Basis modelliert werden,
// da die verwendete Version von Synopsys Behavioral Compiler kein
// Scheduling von Verhaltensbeschreibungen mit 'INOUT'-Ports zulaesst.
// ----------------------------------------------------------------------------

   input             RnW;              // Daten bei RnW=1 nach aussen
                                       // durchschalten (Schreibzugriff)
   input    [15:0]   DATA_WRITE;       // ausgehende Daten
   output   [15:0]   DATA_READ;        // eingehende Daten
   inout    [15:0]   DATA_BIDIRECT;    // bidirektionaler Bus

   reg      [15:0]   DATA_DRV;         // Treiber


   assign DATA_BIDIRECT = DATA_DRV;
   assign DATA_READ = DATA_BIDIRECT;

   always @(RnW or DATA_WRITE) begin
      if (RnW)
         DATA_DRV <= 'bz;        // Lesezugriff -> Ausgang hochohmig
      else
         DATA_DRV <= DATA_WRITE; // Schreibzugriff
   end

endmodule
// ----------------------------------------------------------------------------



module mrisc_core (CLK, nRST, WRITECLK, RnW, ADDR, DATAIN, DATAOUT);
// ----------------------------------------------------------------------------
// Kern des Prozessors mit zwei getrennten unidirektionalen Datenbussen
// fuer die Ein- und Ausgabe
// ----------------------------------------------------------------------------

   input             CLK,        // Takteingang
                     nRST,       // asynchron. Reset-Signal, nicht modelliert!
                                 // wird waehrend der Synthese erzeugt
                     WRITECLK;   // zur Maskierung von RnW
   output            RnW;        // Datenrichtung: Schreiben bei RnW == 0
   output   [15:0]   ADDR;       // externer Adressbus
   input    [15:0]   DATAIN;     // interner unidir. Datenbus: eingehende Daten
   output   [15:0]   DATAOUT;    // ausgehende Daten
   reg               WRITE;      // Schreibzyklus bei WRITE==1
   reg      [15:0]   PC,         // Programmzaehler
                     ADR,        // internes Adressregister
                     ADDR,
                     DATAOUT,
                     OPDATA,     // Datenoperand, zu verschiebendes Datum
                     P1,         // Pointerregister fuer indirekte
                     P2,         //   Adressierung
                     AC;         // ALU-Register: Akkumulator und Flags
   reg               C,N,O,V,Z,P,A15,C15,D15;


   assign RnW = ~(WRITE & WRITECLK);   // externe Schreibfreigabe generieren

   // synopsys translate_off
   initial begin                 // nur fuer Verilog-Simulation benoetigte
      WRITE = 0;                 // Initialisierung
      PC = 16'h0010;
```

Listing B.9: MRISC-Prozessor in High-Level-Verilog nach [Friedr98]

```
      ADDR = 0;
   end
   // synopsys translate_on


   always begin : MRISC_CORE

      // Initialisierungszyklus, 0-Pegel am asynchronen Reset-Signal nRST
      // startet die Verarbeitung hier. nRST wird hier nicht explizit
      // modelliert, Funktionalitaet wird bei der Synthese erzeugt
      PC = 16'h0010;
      P1 = 16'h0000;
      P2 = 16'h0000;
      AC = 16'h0000;
      C  = 1'b0;
      N  = 1'b0;
      O  = 1'b0;
      V  = 1'b0;
      Z  = 1'b0;
      P  = 1'b0;
      WRITE <= 1'b0;
      ADDR <= PC;
      DATAOUT <= 16'h0000;
      @(posedge CLK);

      // synopsys translate_off
      while (!nRST) @(posedge CLK); // minimale Reset-Funktionalitaet, die
                                    // fuer eine Verilog-Simulation noetig ist
      // synopsys translate_on


      forever begin : MAIN_LOOP

         // ----- Quelladresse lesen -----

         ADR = DATAIN;
         case (ADR) // synopsys full_case parallel_case
            16'h0002 : ADDR <= P1;  // indirekte, ...
            16'h0004 : ADDR <= P2;
            default  : ADDR <= ADR; // direkte Adressierung
         endcase

         // synopsys translate_off
                                    // Nur fuer Verilog-Simulation:
         if (PC < 16) $stop;        // HALT bei Programmausf. im int. Speicher
         $display ("PC: %h",PC);    // Programmzaehler ausgeben
         // synopsys translate_on

         PC = PC + 1'b1;
         @(posedge CLK);

         // ----- Operand lesen -----

         case (ADR) // synopsys full_case parallel_case
            16'h0000 : OPDATA = PC + 3;   // interne Register:
            16'h0001 : OPDATA = P1;
            16'h0003 : OPDATA = P2;
            16'h0005 : OPDATA = AC;
            16'h0006 : OPDATA = ~AC;
            16'h0007 : OPDATA = { 14'b0, N, 1'b0 };
            16'h0008 : OPDATA = { 14'b0, O, 1'b0 };
            16'h0009 : OPDATA = { 14'b0, Z, 1'b0 };
            16'h000a : OPDATA = { 14'b0, P, 1'b0 };
            16'h000b : OPDATA = { 14'b0, V, 1'b0 };
            16'h000c : OPDATA = { 14'b0, C, 1'b0 };
            16'h000d : OPDATA = { 14'b0, N^V, 1'b0 };
            16'h000e : OPDATA = { 14'b0, (N^V)|Z, 1'b0 };
            16'h000f : OPDATA = { 14'b0, ~(C|Z), 1'b0 };
            default  : OPDATA = DATAIN;   // externer Speicherzugriff
         endcase
         ADDR <= PC;
         @(posedge CLK);

         // ----- Zieladresse lesen -----
```

Listing B.9: MRISC-Prozessor in High-Level-Verilog nach [Friedr98]

```
        PC = PC + 1'b1;
        ADR = DATAIN;
        case (ADR) // synopsys full_case parallel_case
           16'h0002 : ADDR <= P1;  // indirekte, ...
           16'h0004 : ADDR <= P2;
           default  : ADDR <= ADR; // direkte Adressierung
        endcase
        DATAOUT <= OPDATA;         // Daten ausgeben
        WRITE <= 1'b1;             // Schreibzyklus anzeigen
        @(posedge CLK);

        // ----- Operand schreiben -----

        case (ADR) // synopsys parallel_case
           16'h0000 : PC = OPDATA;
           16'h0001 : P1 = OPDATA;
           16'h0003 : P2 = OPDATA;
           16'h0005 : begin              // Identitaet
                       AC = OPDATA;
                       C = 1'b0;
                       V = 1'b0;
           end
           16'h0006 : begin              // Addition
                       A15 = AC[15];
                       D15 = OPDATA[15];
                       { C, AC } = { 1'b0, AC } + { 1'b0, OPDATA };
                       C15 = AC[15] ^ A15 ^ D15;
                       V = (C ^ C15);    // Ueberlauf rueckrechnen
           end
           16'h0007 : begin              // Subtraktion
                       A15 = AC[15];
                       D15 = OPDATA[15];
                       { C, AC } = { 1'b0, OPDATA } - { 1'b0, AC };
                       C15 = AC[15] ^ ~A15 ^ D15;
                       V = (C ^ ~C15);   // Ueberlauf rueckrechnen
           end
           16'h0008 : begin              // Subtraktion
                       A15 = AC[15];
                       D15 = OPDATA[15];
                       { C, AC } = { 1'b0, AC } - { 1'b0, OPDATA };
                       C15 = AC[15] ^ A15 ^ ~D15;
                       V = (C ^ ~C15);   // Ueberlauf rueckrechnen
           end
           16'h0009 : begin              // bitweises AND
                       AC = AC & OPDATA;
                       C = 1'b0;
                       V = 1'b0;
           end
           16'h000a : begin              // bitweises OR
                       AC = AC | OPDATA;
                       C = 1'b0;
                       V = 1'b0;
           end
           16'h000b : begin              // bitweises XOR
                       AC = AC ^ OPDATA;
                       C = 1'b0;
                       V = 1'b0;
           end
           16'h000c : begin              // Multiplikation mit 2
                       {C,AC} = {OPDATA[15:0],1'b0};
                       V = 1'b0;
           end
           16'h000d : begin              // Division durch 2
                       {AC,C} = {1'b0,OPDATA[15:0]};
                       V = 1'b0;
           end
           16'h000e : begin              // rotiere rechtsherum
                       AC = {OPDATA[0],OPDATA[15:1]};
                       C = OPDATA[0];
                       V = 1'b0;
           end
           16'h000f : begin              // Vorzeichenrichtige Div. durch 2
                       {AC,C} = {OPDATA[15],OPDATA[15:0]};
```

Listing B.9: MRISC-Prozessor in High-Level-Verilog nach [Friedr98]

```
                        V = 1'b0;
            end
         endcase

         if ((ADR >= 16'h004) && (ADR < 16'h0010)) begin    // Flags berechnen
            N = AC[15];
            O = AC[0];
            Z = ~(|AC[15:0]);
            P = ^AC[15:0];
         end

         WRITE <= 1'b0;
         ADDR <= PC;
         @(posedge CLK);

         // ----- Quelladresse lesen -----

      end
   end

endmodule
// ------------------------------------------------------------------------
```

Listing B.9: MRISC-Prozessor in High-Level-Verilog nach [Friedr98]

B.4 Chipkartenleser

B.4.1 Controllersynthese

Das Protocol-Compiler-Modell des Chipkartenlesers, dessen Hierarchie in Bild B.6 dargestellt ist, entspricht in seinen Frame-Definitionen Bild B.7 bis Bild B.13. Die in der Protokolldefinition aufgerufene User-Defined-Action bcd_add zur Addition von Binär codierten Dezimalzahlen ist in Listing B.10 aufgeführt.

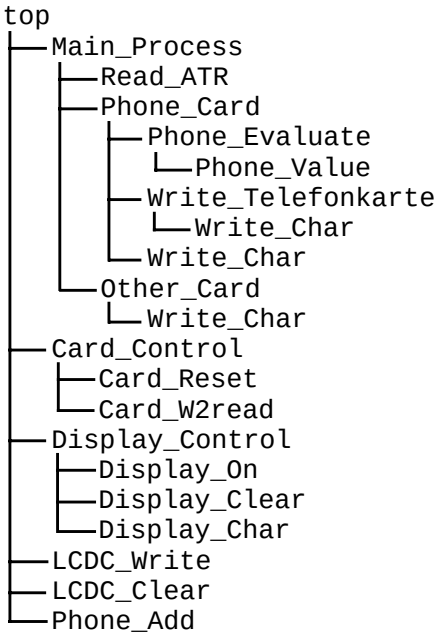


Bild B.6: Frame-Hierarchie des Chipkartenlesers

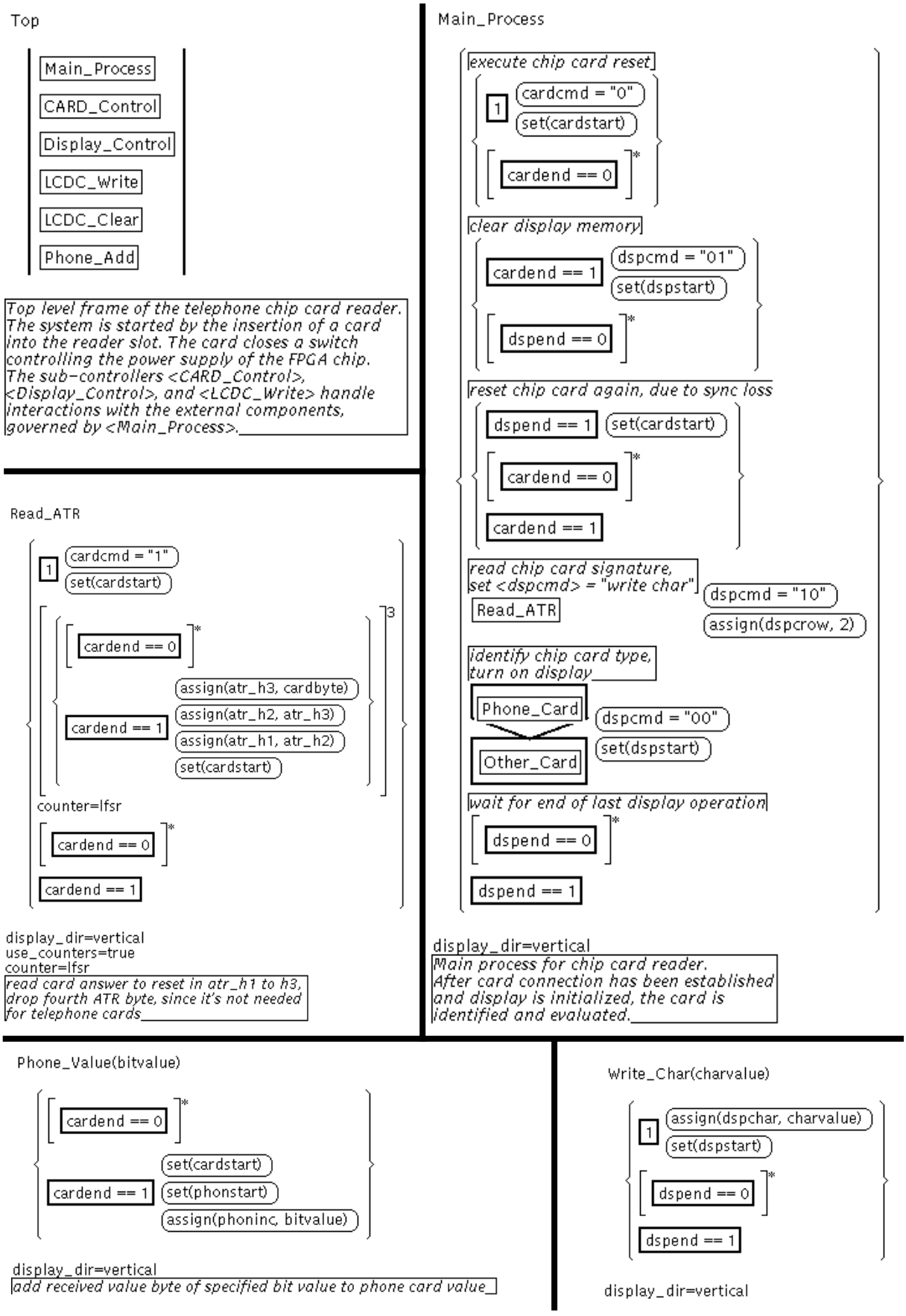
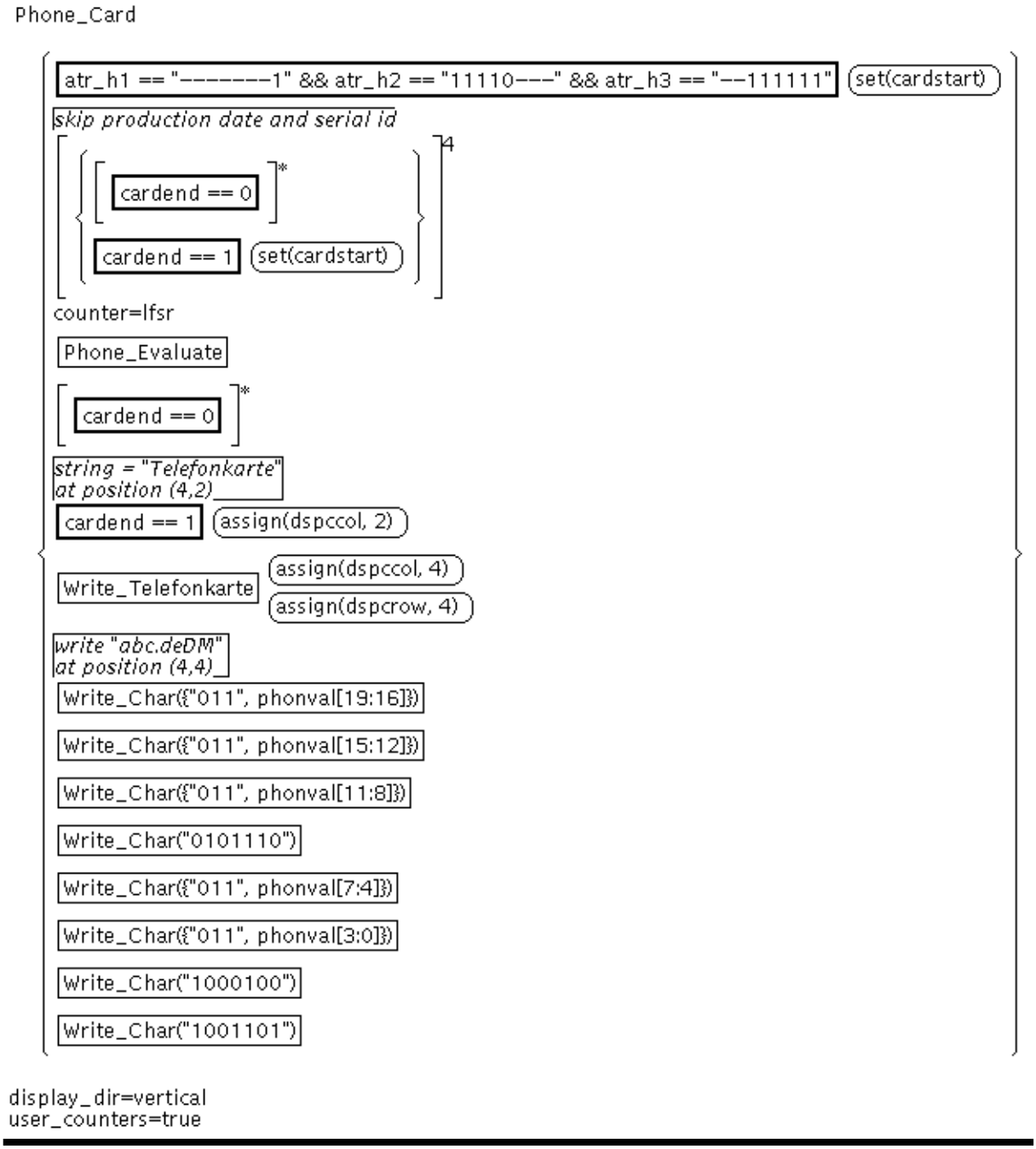


Bild B.7: Protocol-Compiler-Modell des Chipkartenlesers (Teil 1)




```
Ports:
 Clock, in       LCDR, out       ROMA<15:0>, out
 Reset, in       LCDD, out       DBO<7:0>, out
 CRST, out       LCDE, out       DBI<7:0>, in
 CCLK, out       LCD1, out
 CDAT, in        LCD2, out

Functions:
 bcd_add(in bcdval1, in bcdval2) bit_vector<19:0>
```

Bild B.8: Protocol-Compiler-Modell des Chipkartenlesers (Teil 2)

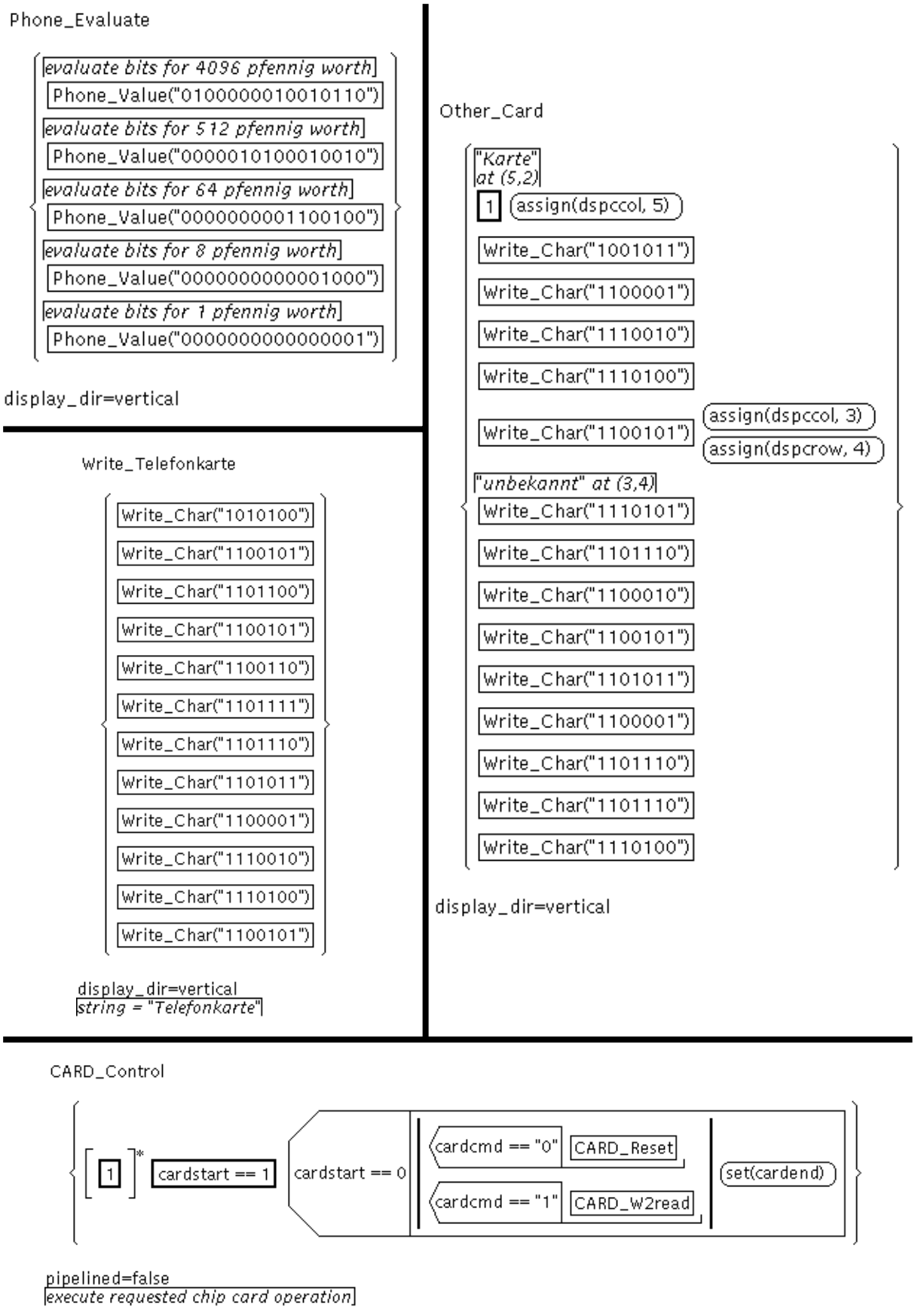
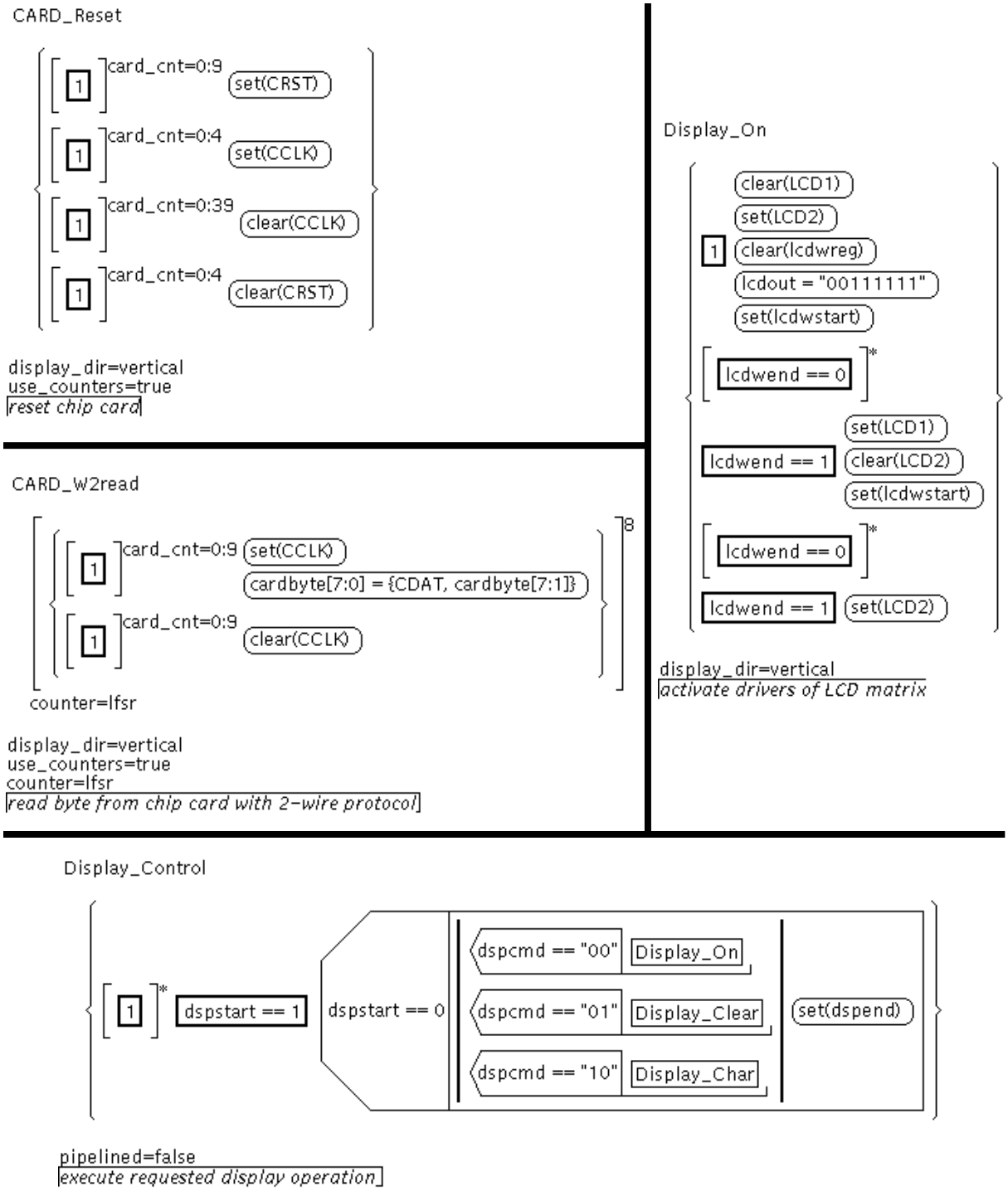


Bild B.9: Protocol-Compiler-Modell des Chipkartenlesers (Teil 3)



Variablen:

cardstart	phonstart	dspstart	lcdstart
cardend	phonval<19:0>	dspend	lcdend
cardcmd	phoninc<19:0>	dspcmd<1:0>	lcdout<7:0>
cardbyte<7:0>	phonbyte<7:0>	dsprow<2:0>	lcdwreg
card_cnt<5:0>		dspcidx<2:0>	lcdwstart
atr_h1<7:0>		dspchar<6:0>	lcdwend
atr_h2<7:0>		dspcrow<2:0>	
atr_h3<7:0>		dspccol<3:0>	

Bild B.10: Protocol-Compiler-Modell des Chipkartenlesers (Teil 4)

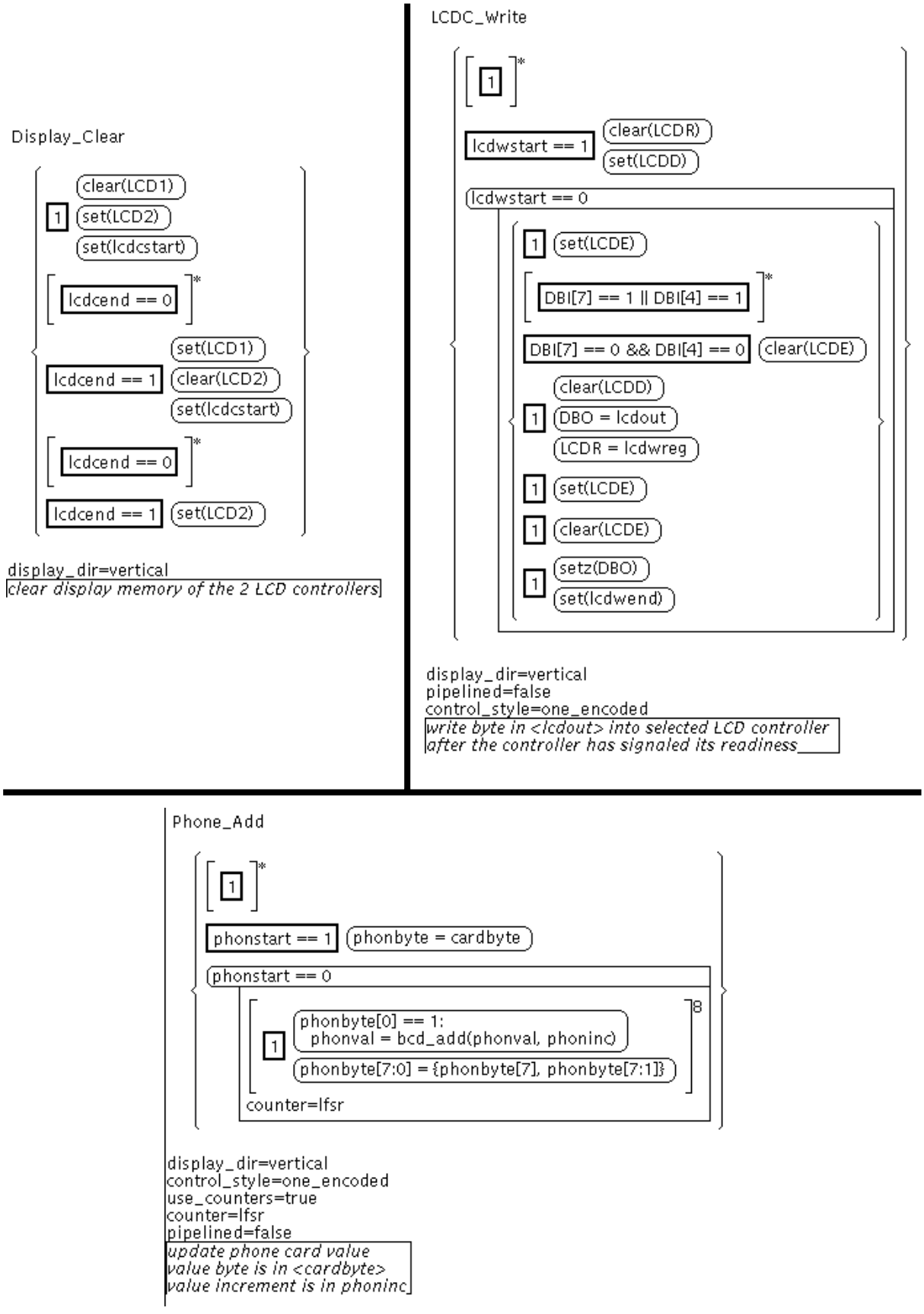
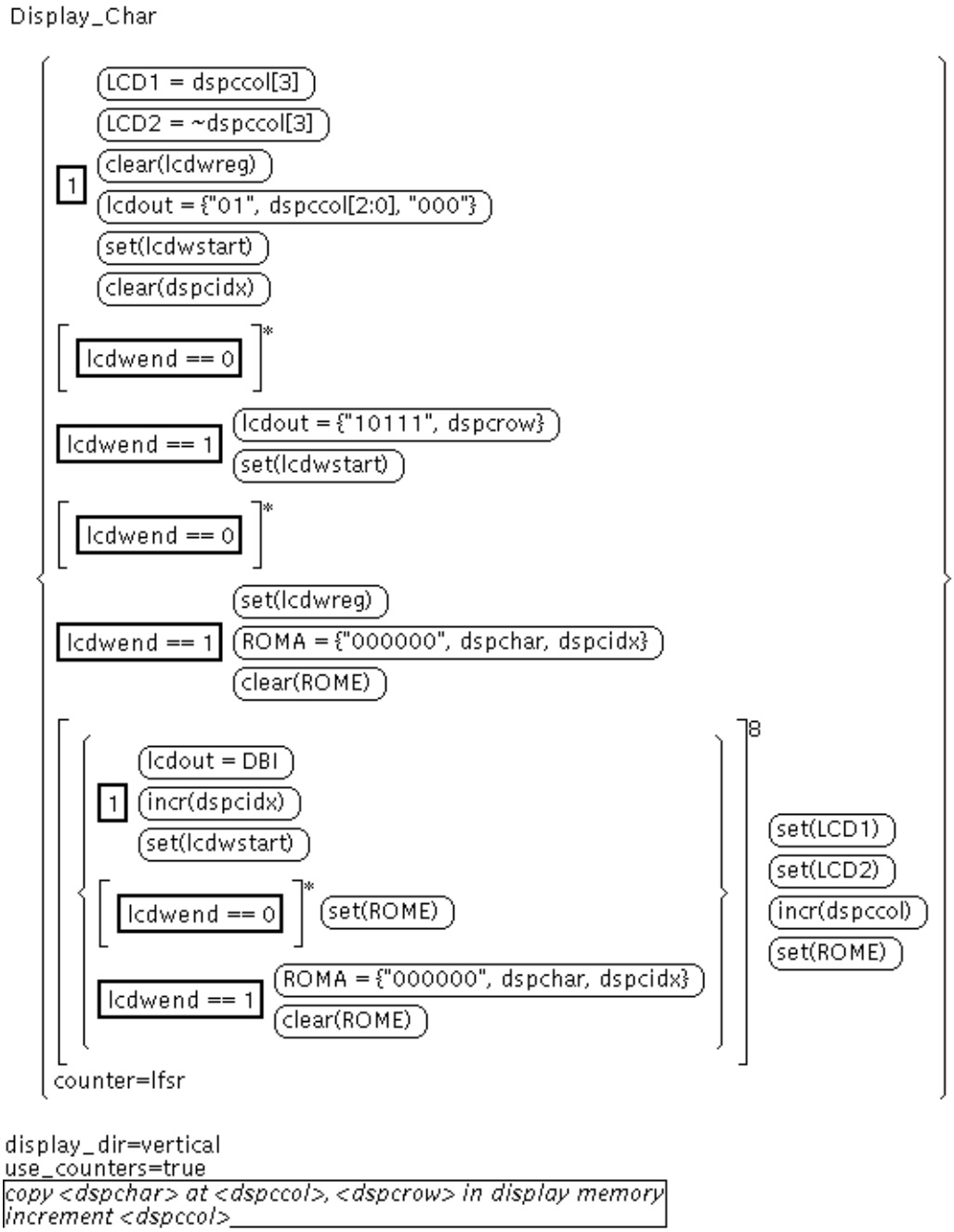


Bild B.11: Protocol-Compiler-Modell des Chipkartenlesers (Teil 5)




```
Reset-Actions:
 clear(phonstart)
 clear(phonval)
 clear(dspstart)
 clear(dspend)
 clear(dsprow)
 clear(lcdstart)
 clear(lcdend)
 clear(lcdwstart)
 clear(lcdwend)
```

```
Default-Actions:
 clear(phonstart)
 clear(dspstart)
 clear(dspend)
 clear(lcdstart)
 clear(lcdend)
 clear(lcdwstart)
 clear(lcdwend)
```

Bild B.12: Protocol-Compiler-Modell des Chipkartenlesers (Teil 6)

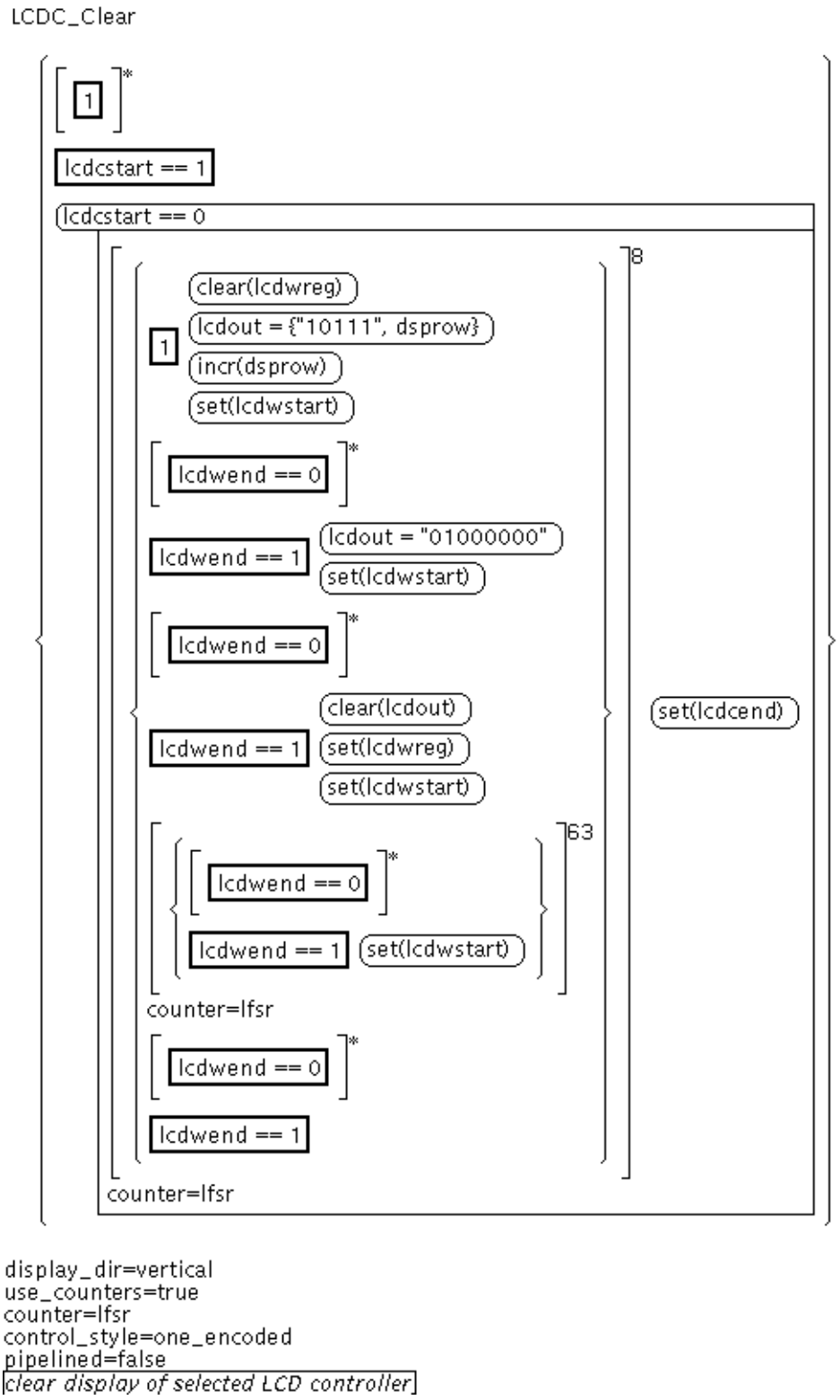


Bild B.13: Protocol-Compiler-Modell des Chipkartenlesers (Teil 7)

```
//
// BCD-Adder fuer 5-stellige BCD-Zahlen
// im Rahmen des Dali-Chipkartenleser-Projektes
// PB990114
//

function[19:0]  bcd_add;
  input  [19:0] bcdval1;
  input  [19:0] bcdval2;
  reg           carry;
  reg    [19:0] temp;
begin
  {carry,temp[3:0]} = bcd_digit_add(bcdval1[3:0],bcdval2[3:0],0);
  {carry,temp[7:4]} = bcd_digit_add(bcdval1[7:4],bcdval2[7:4],carry);
  {carry,temp[11:8]} = bcd_digit_add(bcdval1[11:8],bcdval2[11:8],carry);
  {carry,temp[15:12]} = bcd_digit_add(bcdval1[15:12],bcdval2[15:12],carry);
  {carry,temp[19:16]} = bcd_digit_add(bcdval1[19:16],bcdval2[19:16],carry);
  bcd_add = temp;
end
endfunction

function[4:0] bcd_digit_add;
  input  [3:0] value1;
  input  [3:0] value2;
  input        carry;
  reg    [4:0] sumtmp;
begin
  sumtmp = value1 + value2 + carry;
  case(sumtmp)
     0: bcd_digit_add = 5'b00000;
     1: bcd_digit_add = 5'b00001;
     2: bcd_digit_add = 5'b00010;
     3: bcd_digit_add = 5'b00011;
     4: bcd_digit_add = 5'b00100;
     5: bcd_digit_add = 5'b00101;
     6: bcd_digit_add = 5'b00110;
     7: bcd_digit_add = 5'b00111;
     8: bcd_digit_add = 5'b01000;
     9: bcd_digit_add = 5'b01001;
    10: bcd_digit_add = 5'b10000;
    11: bcd_digit_add = 5'b10001;
    12: bcd_digit_add = 5'b10010;
    13: bcd_digit_add = 5'b10011;
    14: bcd_digit_add = 5'b10100;
    15: bcd_digit_add = 5'b10101;
    16: bcd_digit_add = 5'b10110;
    17: bcd_digit_add = 5'b10111;
    18: bcd_digit_add = 5'b11000;
    19: bcd_digit_add = 5'b11001;
    default: bcd_digit_add = 5'bxxxxx;
  endcase
end
endfunction
```

Listing B.10: User-Defined-Function für BCD-Addition im Chipkartenleser

B.5 IDEA-Datenpfad

B.5.1 RTL-Synthese

In Lisiting B.11 ist das RTL-Modell der untersuchten Kodierungsstufe des IDEA-Datenpfades enthalten.

```
//
// IDEA-Kodierungsstufe in RTL-Verilog
//
module ideastep (OBLOCK, IBLOCK, KEY, CLOCK);
```

Listing B.11: IDEA-Kodierungsstufe in RTL-Verilog

```
  output [63:0] OBLOCK;   // output block
  input  [63:0] IBLOCK;   // input block
  input  [95:0] KEY;      // key code
  input         CLOCK;

  wire   [95:0] KEY;      // key code
  wire   [63:0] IBLOCK;   // input block
  reg    [63:0] OBLOCK;   // output block
  wire          CLOCK;
  wire   [15:0] TEMP_A, TEMP_B, TEMP_C, TEMP_D, TEMP_E, TEMP_F, TEMP_G,
                TEMP_H, TEMP_I, TEMP_J, TEMP_K, TEMP_L, TEMP_M, TEMP_N;

  ideamul MUL_A (TEMP_A,  IBLOCK[63:48],  KEY[95:80]);
  assign         TEMP_B = IBLOCK[47:32] + KEY[79:64];
  assign         TEMP_C = IBLOCK[31:16] + KEY[63:48];
  ideamul MUL_B (TEMP_D,  IBLOCK[15:0],   KEY[47:32]);
  assign         TEMP_E = TEMP_A ^ TEMP_C;
  assign         TEMP_F = TEMP_B ^ TEMP_D;
  ideamul MUL_C (TEMP_G,  TEMP_E,  KEY[31:16]);
  assign         TEMP_H = TEMP_G + TEMP_F;
  ideamul MUL_D (TEMP_I,  TEMP_H,  KEY[15:0]);
  assign         TEMP_J = TEMP_G + TEMP_I;
  assign         TEMP_K = TEMP_A ^ TEMP_I;
  assign         TEMP_L = TEMP_C ^ TEMP_I;
  assign         TEMP_M = TEMP_B ^ TEMP_J;
  assign         TEMP_N = TEMP_D ^ TEMP_J;

  always @(posedge CLOCK) begin
    OBLOCK <= {TEMP_K, TEMP_L, TEMP_M, TEMP_N};
  end

endmodule

//
// IDEA-Multiplikation (Multiplikation Modulo 2^16+1)
//
module ideamul(result, operand1, operand2);

  output [15:0] result;
  input  [15:0] operand1,
                operand2;

  reg    [15:0] result;
  wire   [15:0] operand1,
                operand2;

  reg    [31:0] mulreg;
  reg    [15:0] offset;

  always @(operand1 or operand2) begin
    if (|operand1 == 0) result = 1 - operand2;
    else if (|operand2 == 0) result = 1 - operand1;
         else begin
           mulreg = operand1 * operand2;
           offset = mulreg[15:0] - mulreg[31:16];
           if (mulreg[15:0] <= mulreg[31:16]) result = offset + 1;
           else                               result = offset;
         end
  end

endmodule
```

Listing B.11: IDEA-Kodierungsstufe in RTL-Verilog

B.5.2 High-Level-Synthese

Das in den Untersuchungen betrachtete High-Level-Modell der IDEA-Kodierungsstufe zeigt Listing B.12.

```
//
// IDEA-Kodierungsstufe in High-Level-Verilog
//
```

Listing B.12: IDEA-Kodierungsstufe in High-Level-Verilog

```
module ideastep (OBLOCK, IBLOCK, KEY, CLOCK);

  output [63:0] OBLOCK;   // output block
  input  [63:0] IBLOCK;   // input block
  input  [95:0] KEY;      // key code
  input         CLOCK;

  wire   [95:0] KEY;      // key code
  wire   [63:0] IBLOCK;   // input block
  reg    [63:0] OBLOCK;   // output block
  wire          CLOCK;
  reg    [15:0] TEMP_A, TEMP_B, TEMP_C, TEMP_D, TEMP_E, TEMP_F, TEMP_G,
                TEMP_H, TEMP_I, TEMP_J, TEMP_K, TEMP_L, TEMP_M, TEMP_N;

  always begin
    @(posedge CLOCK);
    TEMP_A = ideamul(IBLOCK[63:48],  KEY[95:80]);
    TEMP_B = IBLOCK[47:32] + KEY[79:64];
    TEMP_C = IBLOCK[31:16] + KEY[63:48];
    TEMP_D = ideamul(IBLOCK[15:0],   KEY[47:32]);
    TEMP_E = TEMP_A ^ TEMP_C;
    TEMP_F = TEMP_B ^ TEMP_D;
    TEMP_G = ideamul(TEMP_E,  KEY[31:16]);
    TEMP_H = TEMP_G + TEMP_F;
    TEMP_I = ideamul(TEMP_H,  KEY[15:0]);
    TEMP_J = TEMP_G + TEMP_I;
    TEMP_K = TEMP_A ^ TEMP_I;
    TEMP_L = TEMP_C ^ TEMP_I;
    TEMP_M = TEMP_B ^ TEMP_J;
    TEMP_N = TEMP_D ^ TEMP_J;
    OBLOCK <= {TEMP_K, TEMP_L, TEMP_M, TEMP_N};
  end


  function [15:0] ideamul;
    input  [15:0] operand1,
                  operand2;
    reg    [31:0] mulreg;
    reg    [15:0] offset;

    begin
      if (|operand1 == 0) ideamul = 1 - operand2;
      else if (|operand2 == 0) ideamul = 1 - operand1;
           else begin
             mulreg = operand1 * operand2;
             offset = mulreg[15:0] - mulreg[31:16];
             if (mulreg[15:0] <= mulreg[31:16]) ideamul = offset + 1;
             else                               ideamul = offset;
           end
    end
 endfunction

endmodule
```

Listing B.12: IDEA-Kodierungsstufe in High-Level-Verilog